Hans Kellerer
Ulrich Pferschy
David Pisinger

Knapsack Problems



Springer

Knapsack Problems

Springer-Verlag Berlin Heidelberg GmbH

Hans Kellerer · Ulrich Pferschy
David Pisinger

Knapsack Problems

With 105 Figures
and 33 Tables



Springer

Prof. Hans Kellerer
University of Graz
Department of Statistics and Operations Research
Universitätsstr. 15
A-8010 Graz, Austria
hans.kellerer@uni-graz.at

Prof. Ulrich Pferschy
University of Graz
Department of Statistics and Operations Research
Universitätsstr. 15
A-8010 Graz, Austria
pferschy@uni-graz.at

Prof. David Pisinger
University of Copenhagen
DIKU, Department of Computer Science
Universitetsparken 1
DK-2100 Copenhagen, Denmark
pisinger@diku.dk

ISBN 978-3-642-07311-3 ISBN 978-3-540-24777-7 (eBook)
DOI 10.1007/978-3-540-24777-7

Cataloging-in-Publication Data applied for
A catalog record for this book is available from the Library of Congress.
Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

springeronline.com

© Springer-Verlag Berlin Heidelberg 2004
Originally published by Springer-Verlag Berlin Heidelberg New York in 2004
Softcover reprint of the hardcover 1st edition 2004

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: Erich Kirchner, Heidelberg

Preface

Thirteen years have passed since the seminal book on knapsack problems by Martello and Toth appeared. On this occasion a former colleague exclaimed back in 1990: "How can you write 250 pages on the knapsack problem?" Indeed, the definition of the knapsack problem is easily understood even by a non-expert who will not suspect the presence of challenging research topics in this area at the first glance.

However, in the last decade a large number of research publications contributed new results for the knapsack problem in all areas of interest such as exact algorithms, heuristics and approximation schemes. Moreover, the extension of the knapsack problem to higher dimensions both in the number of constraints and in the number of knapsacks, as well as the modification of the problem structure concerning the available item set and the objective function, leads to a number of interesting variations of practical relevance which were the subject of intensive research during the last few years.

Hence, two years ago the idea arose to produce a new monograph covering not only the most recent developments of the standard knapsack problem, but also giving a comprehensive treatment of the whole knapsack family including the siblings such as the subset sum problem and the bounded and unbounded knapsack problem, and also more distant relatives such as multidimensional, multiple, multiple-choice and quadratic knapsack problems in dedicated chapters.

Furthermore, attention is paid to a number of less frequently considered variants of the knapsack problem and to the study of stochastic aspects of the problem. To illustrate the high practical relevance of the knapsack family for many industrial and economic problems, a number of applications are described in more detail. They are selected subjectively from the innumerable occurrences of knapsack problems reported in the literature.

Our above-mentioned colleague will be surprised to notice that even on the more than 500 pages of this book not all relevant topics could be treated in equal depth but decisions had to be made on where to go into details of constructions and proofs and where to concentrate on stating results and refer to the appropriate publications. Moreover, an editorial deadline had to be drawn at some point. In our case, we stopped looking for new publications at the end of June 2003.

The audience we envision for this book is threefold: The first two chapters offer a very basic introduction to the knapsack problem and the main algorithmic concepts to derive optimal and approximate solution. Chapter 3 presents a number of advanced algorithmic techniques which are used throughout the later chapters of the book. The style of presentation in these three chapters is kept rather simple and assumes only minimal prerequisites. They should be accessible to students and graduates of business administration, economics and engineering as well as practitioners with little knowledge of algorithms and optimization.

This first part of the book is also well suited to introduce classical concepts of optimization in a classroom, since the knapsack problem is easy to understand and is probably the least difficult but most illustrative problem where dynamic programming, branch-and-bound, relaxations and approximation schemes can be applied.

In these chapters no knowledge of linear or integer programming and only a minimal familiarity with basic elements of graph theory is assumed. The issue of $\mathcal{NP}$-completeness is dealt with by an intuitive introduction in Section 1.5, whereas a thorough and rigorous treatment is deferred to the Appendix.

The remaining chapters of the book address two different audiences. On one hand, a student or graduate of mathematics or computer science, or a successful reader of the first three chapters willing to go into more depth, can use this book to study advanced algorithms for the knapsack problem and its relatives. On the other hand, we hope scientific researchers or expert practitioners will find the book a valuable source of reference for a quick update on the state of the art and on the most efficient algorithms currently available. In particular, a collection of computational experiments, many of them published for the first time in this book, should serve as a valuable tool to pick the algorithm best suited for a given problem instance. To facilitate the use of the book as a reference we tried to keep these chapters self-contained as far as possible.

For these advanced audiences we assume familiarity with the basic theory of linear programming, elementary elements of graph theory, and concepts of algorithms and data structures as far as they are generally taught in basic courses on these subjects.

Chapters 4 to 12 give detailed presentations of the knapsack problem and its variants in increasing order of structural difficulty. Hence, we start with the subset sum problem in Chapter 4, move on to the standard knapsack problem which is discussed extensively in two chapters, one for exact and one for approximate algorithms, and finish this second part of the book with the bounded and unbounded knapsack problem in Chapters 7 and 8.

The third part of the book contains more complicated generalizations of the knapsack problems. It starts with the multidimensional knapsack problem (a knapsack problem with d constraints) in Chapter 9, then considers the multiple knapsack problem (m knapsacks are available for packing) in Chapter 10, goes on to the multiple-choice knapsack problem (the items are partitioned into classes and exactly one item of each class must be packed), and extends the linear objective func-

tion to a quadratic one yielding the quadratic knapsack problem in Chapter 12. This chapter also contains an excursion to semidefinite programming giving a mostly self-contained short introduction to this topic.

A collection of other variants of the knapsack problem is put together in Chapter 13. Detailed expositions are devoted to the multiobjective and the precedence constraint knapsack problem, whereas other subjectively selected variants are treated in a more cursory way. The solitary Chapter 14 gives a survey on stochastic results for the knapsack problem. It also contains a section on the on-line version of the problem.

All these six chapters can be seen as survey articles, most of them being the first survey on their subject, containing many pointers to the literature and some examples of application.

Particular effort was put into the description of interesting applications of knapsack type problems. We decided to avoid a boring listing of umpteen papers with a two-line description of the occurrence of a knapsack problem for each of them, but selected a smaller number of application areas where knapsack models play a prominent role. These areas are discussed in more detail in Chapter 15 to give the reader a full understanding of the situations presented. They should be particularly useful for teaching purposes.

The Appendix gives a short presentation of $\mathcal{NP}$-completeness with the focus on knapsack problems. Without venturing into the depths of theoretical computer science and avoiding topics such as Turing machines and unary encoding, a rather informal introduction to $\mathcal{NP}$-completeness is given, however with formal proofs for the $\mathcal{NP}$-hardness of the subset sum and the knapsack problem.

Some assumptions and conventions concerning notation and style are kept throughout the book. Most algorithms are stated in a flexible pseudocode style putting emphasis on readability instead of formal uniformity. This means that simpler algorithms are given in the style of an unknown but easily understandable programming language, whereas more complex algorithms are introduced by a structured, but verbal description. Commands and verbal instructions are given in Sans Serif font, whereas comments follow in *Italic* letters. As a general reference and guideline to algorithms we used the book by Cormen, Leiserson, Rivest and Stein [92].

For the sake of readability and personal taste we follow the non-standard convention of using the term *increasing* instead of the mathematically correct *nondecreasing* and in the same way *decreasing* instead of *nonincreasing*. Whereever we use the log function we always refer to the base 2 logarithm unless stated otherwise. After the Preface we give a short list of notations containing only those terms which are used throughout the book. Many more naming conventions will be introduced on a local level during the individual chapters and sections.

As mentioned above a number of computational experiments were performed for exact algorithms. These were performed on the following machines:

AMD ATHLON, 1.2 GHz	SPECint2000 = 496	SPECfp2000 = 417
INTEL PENTIUM 4, 1.5 GHz	SPECint2000 = 558	SPECfp2000 = 615
INTEL PENTIUM III, 933 MHz	SPECint2000 = 403	SPECfp2000 = 328

The performance index was obtained from SPEC (`www.specbench.org`). As can be seen the three machines have reasonably similar performance, making it possible to compare running times across chapters. The codes have been compiled using the GNU project C and C++ Compiler gcc-2.96, which also compiles Fortran77 code, thus preventing differences in computation times due to alternative compilers.

Acknowledgements

The authors strongly believe in the necessity to do research not with an island mentality but in an open exchange of knowledge, opinions and ideas within an international research community. Clearly, none of us would have been able to contribute to this book without the innumerable personal exchanges with colleagues on conferences and workshops, in person, by e-mail or even by surface mail. Therefore, we would like to start our acknowledgements by thanking the global research community for providing the spirit necessary for joint projects of collection and presentation.

The classic book by Silvano Martello and Paolo Toth on knapsack problems was frequently used as a reference during the writing of this text. Comments by both authors were greatly appreciated.

To our personal friends Alberto Caprara and Eranda Cela we owe special thanks for many discussions and helpful suggestions. John M. Bergstrom brought to our attention the importance of solving knapsack problems for all values of the capacity. Jarl Friis gave valuable comments on the chapter on the quadratic knapsack problem. Klaus Ladner gave valuable technical support, in particular in the preparation of figures.

In the computational experiments and comparisons, codes were used which were made available by Martin E. Dyer, Silvano Martello, Nei Y. Soma, Paolo Toth and John Walker. We thank them for their cooperation. Anders Bo Rasmussen and Rune Sandvik deserve special thanks for having implemented the upper bounds for the quadratic knapsack problem in Chapter 12 and for having run the computational experiments with these bounds. In this context the authors would also like to acknowledge DIKU Copenhagen for having provided the computational facilities for the computational experiments.

Finally, we would like to thank the Austrian and Danish tax payer for enabling us to devote most of our concentration on the writing of this book during the last two years. We would also like to apologize to the colleagues of our departments, our friends and our families for having neglected them during this time. Further apologies go to the reader of this book for any errors and mistakes it contains. These will be collected at the web-site of this book at `www.diku.dk/knapsack`.

Table of Contents

Preface V

Table of Contents IX

List of Notations XIX

1. Introduction 1

1.1 Introducing the Knapsack Problem 1

1.2 Variants and Extensions of the Knapsack Problem 5

1.3 Single-Capacity Versus All-Capacities Problem 9

1.4 Assumptions on the Input Data 9

1.5 Performance of Algorithms 11

2. Basic Algorithmic Concepts 15

2.1 The Greedy Algorithm 15

2.2 Linear Programming Relaxation 17

2.3 Dynamic Programming 20

2.4 Branch-and-Bound 27

2.5 Approximation Algorithms 29

2.6 Approximation Schemes 37

3. Advanced Algorithmic Concepts 43
3.1 Finding the Split Item in Linear Time 43
3.2 Variable Reduction 44
3.3 Storage Reduction in Dynamic Programming 46
3.4 Dynamic Programming with Lists 50
3.5 Combining Dynamic Programming and Upper Bounds 53
3.6 Balancing 54
3.7 Word RAM Algorithms 60
3.8 Relaxations 62
3.9 Lagrangian Decomposition 65
3.10 The Knapsack Polytope 67

4. The Subset Sum Problem 73
4.1 Dynamic Programming 75
4.1.1 Word RAM Algorithm 76
4.1.2 Primal-Dual Dynamic Programming Algorithms 79
4.1.3 Primal-Dual Word-RAM Algorithm 80
4.1.4 Horowitz and Sahni Decomposition 81
4.1.5 Balancing 82
4.1.6 Bellman Recursion in Decision Form 85
4.2 Branch-and-Bound 85
4.2.1 Upper Bounds 86
4.2.2 Hybrid Algorithms 87
4.3 Core Algorithms 88
4.3.1 Fixed Size Core 89
4.3.2 Expanding Core 89
4.3.3 Fixed Size Core and Decomposition 90
4.4 Computational Results: Exact Algorithms 90
4.4.1 Solution of All-Capacities Problems 93
4.5 Polynomial Time Approximation Schemes for Subset Sum 94
4.6 A Fully Polynomial Time Approximation Scheme for Subset Sum 97
4.7 Computational Results: FPTAS 112

5. Exact Solution of the Knapsack Problem 117
5.1 Branch-and-Bound 119
5.1.1 Upper Bounds for (KP) 119
5.1.2 Lower Bounds for (KP) 124
5.1.3 Variable Reduction 125
5.1.4 Branch-and-Bound Implementations 127
5.2 Primal Dynamic Programming Algorithms 130
5.2.1 Word RAM Algorithm 131
5.2.2 Horowitz and Sahni Decomposition 136
5.3 Primal-Dual Dynamic Programming Algorithms 136
5.3.1 Balanced Dynamic Programming 138
5.4 The Core Concept 140
5.4.1 Finding a Core 142
5.4.2 Core Algorithms 144
5.4.3 Combining Dynamic Programming with Tight Bounds ... 147
5.5 Computational Experiments 150
5.5.1 Difficult Instances 154
5.5.2 Difficult Instances with Large Coefficients 155
5.5.3 Difficult Instances With Small Coefficients 156

6. Approximation Algorithms for the Knapsack Problem 161
6.1 Polynomial Time Approximation Schemes 161
6.1.1 Improving the *PTAS* for (KP) 161
6.2 Fully Polynomial Time Approximation Schemes 166
6.2.1 Scaling and Reduction of the Item Set 169
6.2.2 An Auxiliary Vector Merging Problem 171
6.2.3 Solving the Reduced Problem 175
6.2.4 Putting the Pieces Together 177

7. The Bounded Knapsack Problem 185
7.1 Introduction 185
7.1.1 Transformation of (BKP) into (KP) 187
7.2 Dynamic Programming 190
7.2.1 A Minimal Algorithm for (BKP) 191
7.2.2 Improved Dynamic Programming: Reaching (KP) Complexity for (BKP) 194
7.2.3 Word RAM Algorithm 200
7.2.4 Balancing 200
7.3 Branch-and-Bound 201
7.3.1 Upper Bounds 201
7.3.2 Branch-and Bound Algorithms 202
7.3.3 Computational Experiments 204
7.4 Approximation Algorithms 205

8. The Unbounded Knapsack Problem 211
8.1 Introduction 211
8.2 Periodicity and Dominance 214
8.2.1 Periodicity 215
8.2.2 Dominance 216
8.3 Dynamic Programming 219
8.3.1 Some Basic Algorithms 220
8.3.2 An Advanced Algorithm 223
8.3.3 Word RAM Algorithm 227
8.4 Branch-and-Bound 228
8.5 Approximation Algorithms 232

9. Multidimensional Knapsack Problems 235
9.1 Introduction 235
9.2 Relaxations and Reductions 238
9.3 Exact Algorithms 246
9.3.1 Branch-and-Bound Algorithms 246

9.3.2 Dynamic Programming 248
9.4 Approximation 252
9.4.1 Negative Approximation Results 252
9.4.2 Polynomial Time Approximation Schemes 254
9.5 Heuristic Algorithms 255
9.5.1 Greedy-Type Heuristics 256
9.5.2 Relaxation-Based Heuristics 261
9.5.3 Advanced Heuristics 264
9.5.4 Approximate Dynamic Programming 266
9.5.5 Metaheuristics 268
9.6 The Two-Dimensional Knapsack Problem 269
9.7 The Cardinality Constrained Knapsack Problem 271
9.7.1 Related Problems 272
9.7.2 Branch-and-Bound 273
9.7.3 Dynamic Programming 273
9.7.4 Approximation Algorithms 276
9.8 The Multidimensional Multiple-Choice Knapsack Problem 280

10. Multiple Knapsack Problems 285
10.1 Introduction 285
10.2 Upper Bounds 288
10.2.1 Variable Reduction and Tightening of Constraints 291
10.3 Branch-and-Bound 292
10.3.1 The MTM Algorithm 293
10.3.2 The Mulknap Algorithm 294
10.3.3 Computational Results 296
10.4 Approximation Algorithms 298
10.4.1 Greedy-Type Algorithms and Further Approximation Algorithms 299
10.4.2 Approximability Results for (B-MSSP) 301
10.5 Polynomial Time Approximation Schemes 304
10.5.1 A *PTAS* for the Multiple Subset Problem 304

10.5.2 A *PTAS* for the Multiple Knapsack Problem 311
10.6 Variants of the Multiple Knapsack Problem 315
10.6.1 The Multiple Knapsack Problem with Assignment Restrictions 315
10.6.2 The Class-Constrained Multiple Knapsack Problem 315

11. The Multiple-Choice Knapsack Problem 317
11.1 Introduction 317
11.2 Dominance and Upper Bounds 319
11.2.1 Linear Time Algorithms for the LP-Relaxed Problem 322
11.2.2 Bounds from Lagrangian Relaxation 325
11.2.3 Other Bounds 327
11.3 Class Reduction 327
11.4 Branch-and-Bound 328
11.5 Dynamic Programming 329
11.6 Reduction of States 331
11.7 Hybrid Algorithms and Expanding Core Algorithms 332
11.8 Computational Experiments 335
11.9 Heuristics and Approximation Algorithms 338
11.10 Variants of the Multiple-Choice Knapsack Problem 339
11.10.1 Multiple-Choice Subset Sum Problem 339
11.10.2 Generalized Multiple-Choice Knapsack Problem 340
11.10.3 The Knapsack Sharing Problem 342

12. The Quadratic Knapsack Problem 349
12.1 Introduction 349
12.2 Upper Bounds 351
12.2.1 Continuous Relaxation 352
12.2.2 Bounds from Lagrangian Relaxation of the Capacity Constraint 352
12.2.3 Bounds from Upper Planes 355
12.2.4 Bounds from Linearisation 356

12.2.5 Bounds from Reformulation 359
12.2.6 Bounds from Lagrangian Decomposition 362
12.2.7 Bounds from Semidefinite Relaxation 367
12.3 Variable Reduction 373
12.4 Branch-and-Bound 374
12.5 The Algorithm by Caprara, Pisinger and Toth 375
12.6 Heuristics 379
12.7 Approximation Algorithms 380
12.8 Computational Experiments — Exact Algorithms 382
12.9 Computational Experiments — Upper Bounds 384

13. Other Knapsack Problems 389
13.1 Multiobjective Knapsack Problems 389
13.1.1 Introduction 389
13.1.2 Exact Algorithms for (MOKP) 391
13.1.3 Approximation of the Multiobjective Knapsack Problem . 393
13.1.4 An *FPTAS* for the Multiobjective Knapsack Problem 395
13.1.5 A *PTAS* for (MOd-KP) 397
13.1.6 Metaheuristics 401
13.2 The Precedence Constraint Knapsack Problem (PCKP) 402
13.2.1 Dynamic Programming Algorithms for Trees 404
13.2.2 Other Results for (PCKP) 407
13.3 Further Variants 408
13.3.1 Nonlinear Knapsack Problems 409
13.3.2 The Max-Min Knapsack Problem 411
13.3.3 The Minimization Knapsack Problem 412
13.3.4 The Equality Knapsack Problem 413
13.3.5 The Strongly Correlated Knapsack Problem 414
13.3.6 The Change-Making Problem 415
13.3.7 The Collapsing Knapsack Problem 416
13.3.8 The Parametric Knapsack Problem 419
13.3.9 The Fractional Knapsack Problem 421

13.3.10 The Set-Union Knapsack Problem 423
13.3.11 The Multiperiod Knapsack Problem 424

14. Stochastic Aspects of Knapsack Problems 425
14.1 The Probabilistic Model 426
14.2 Structural Results 427
14.3 Algorithms with Expected Performance Guarantee 430
14.3.1 Related Models and Algorithms 431
14.4 Expected Performance of Greedy-Type Algorithms 433
14.5 Algorithms with Expected Running Time 436
14.6 Results for the Subset Sum Problem 437
14.7 Results for the Multidimensional Knapsack Problem 440
14.8 The On-Line Knapsack Problem 442
14.8.1 Time Dependent On-Line Knapsack Problems 445

15. Some Selected Applications 449
15.1 Two-Dimensional Two-Stage Cutting Problems 449
15.1.1 Cutting a Given Demand from a Minimal Number of Sheets 450
15.1.2 Optimal Utilization of a Single Sheet 452
15.2 Column Generation in Cutting Stock Problems 455
15.3 Separation of Cover Inequalities 459
15.4 Financial Decision Problems 461
15.4.1 Capital Budgeting 461
15.4.2 Portfolio Selection 462
15.4.3 Interbank Clearing Systems 464
15.5 Asset-Backed Securitization 465
15.5.1 Introducing Securitization and Amortization Variants 466
15.5.2 Formal Problem Definition 468
15.5.3 Approximation Algorithms 469
15.6 Knapsack Cryptosystems 472
15.6.1 The Merkle-Hellman Cryptosystem 473

15.6.2 Breaking the Merkle-Hellman Cryptosystem 475
15.6.3 Further Results on Knapsack Cryptosystems 477
15.7 Combinatorial Auctions 478
15.7.1 Multi-Unit Combinatorial Auctions and Multi-Dimensional Knapsacks 479
15.7.2 A Multi-Unit Combinatorial Auction Problem with Decreasing Costs per Unit 481

A. Introduction to $\mathcal{NP}$-Completeness of Knapsack Problems 483
A.1 Definitions 483
A.2 $\mathcal{NP}$-Completeness of the Subset Sum Problem 487
A.2.1 Merging of Constraints 488
A.2.2 $\mathcal{NP}$-Completeness 490
A.3 $\mathcal{NP}$-Completeness of the Knapsack Problem 491
A.4 $\mathcal{NP}$-Completeness of Other Knapsack Problems 491

References 495

Author Index 527

Subject Index 535

List of Notations

n	number of items (jobs)
$N = \{1,\ldots,n\}$	set of items
I	instance
p_j	profit of item j
p_{ij}	profit of item j in knapsack i
w_j	weight of item j
w_{ij}	weight of item j in knapsack i
b_j	upper bound on the number of copies of item type j
c	capacity of a single knapsack
m	number of knapsacks
c_i	capacity of knapsack i
$w(S)$	weight of item set S
$p(S)$	profit of item set S
$c(M) := \sum_{i\in M} c_i$	total capacity of knapsacks in set M
$p_{\max}$	$\max\{p_j \mid j = 1,\ldots,n\}$
$p_{\min}$	$\min\{p_j \mid j = 1,\ldots,n\}$
$w_{\max}$	$\max\{w_j \mid j = 1,\ldots,n\}$
$w_{\min}$	$\min\{w_j \mid j = 1,\ldots,n\}$
$b_{\max}$	$\max\{b_j \mid j = 1,\ldots,n\}$
$c_{\max}$	$\max\{c_i \mid i = 1,\ldots,m\}$
$c_{\min}$	$\min\{c_i \mid i = 1,\ldots,m\}$
$x^* = (x_1^*,\ldots,x_n^*)$	optimal solution vector
z^*	optimal solution value
z^H	solution value for heuristic H
X^*	optimal solution set
X^H	solution set for heuristic H
$z^*(I), z^H(I)$	optimal (resp. heuristic) solution value for instance I
z_S^*	optimal solution to subproblem S
s	split item
$\hat{x}$	split solution
z^{LP}, x^{LP}	solution value (solution vector) of the LP relaxation
$e_j := \frac{p_j}{w_j}$	efficiency of item j
U	upper bound
z^ℓ	lower bound

$KP_j(d)$	knapsack problem with items $\{1,\ldots,j\}$ and capacity d
$z_j(d)$	optimal solution value for $KP_j(d)$
$X_j(d)$	optimal solution set for $KP_j(d)$
$z(d)$	optimal solution value for $KP_n(d)$
$X(d)$	optimal solution set for $KP_n(d)$
$PTAS$	polynomial time approximation scheme
$FPTAS$	fully polynomial time approximation scheme
$(\overline{w},\overline{p})$	state with weight $\overline{w}$ and profit $\overline{p}$
$\oplus$	componentwise addition of lists
W	word size
$C(P)$	linear programming relaxation of problem P
$\lambda=(\lambda_1,\ldots,\lambda_m)$	vector of Lagrangian multipliers
$L(P,\lambda)$	Lagrangian relaxation of problem P
$LD(P)$	Lagrangian dual problem
$\mu=(\mu_1,\ldots,\mu_m)$	vector of surrogate multipliers
$S(P,\mu)$	surrogate relaxation of problem P
$SD(P)$	surrogate dual problem
$\mathrm{conv}(S)$	convex hull of set S
$\dim(S)$	dimension of set S
$C:=\{a,\ldots,b\}$	core of a problem
z_C^*	optimal solution of the core problem
$\mathbb{N}$	the natural numbers $1,2,3,\ldots$
$\mathbb{N}_0$	the numbers $0,1,2,3,\ldots$
$\mathbb{R}$	the real numbers
$\log a$	base 2 logarithm of a
$a\|b$	a is a divisor of b
$\gcd(a,b)$	greatest common divisor of a and b
$\mathrm{lcm}(a,b)$	least common multiple of a and b
$a\equiv b\,(\mathrm{mod}\,m)$	$\exists$ integer λ such that $a=\lambda m+b$
$O(f)$	$\{g(x)\mid \exists c,x_0>0 \text{ s.t. } 0\le g(x)\le cf(x)\ \forall x\ge x_0\}$
$\Theta(f)$	$\{g(x)\mid \exists c_1,c_2,x_0>0 \text{ s.t. } 0\le c_1f(x)\le g(x)\le c_2f(x)\ \forall x\ge x_0\}$

1. Introduction

1.1 Introducing the Knapsack Problem

Every aspect of human life is crucially determined by the result of decisions. Whereas private decisions may be based on emotions or personal taste, the complex professional environment of the 21st century requires a decision process which can be formalized and validated independently from the involved individuals. Therefore, a quantitative formulation of all factors influencing a decision and also of the result of the decision process is sought.

In order to meet this goal it must be possible to represent the effect of any decision by numerical values. In the most basic case the outcome of the decision can be measured by a single value representing gain, profit, loss, cost or some other category of data. The comparison of these values induces a total order on the set of all options which are available for a decision. Finding the option with the highest or lowest value can be difficult because the set of available options may be extremely large and/or not explicitly known. Frequently, only conditions are known which characterize the feasible options out of a very general ground set of theoretically available choices.

The simplest possible form of a decision is the choice between two alternatives. Such a *binary decision* is formulated in a quantitative model as a *binary variable* $x \in \{0,1\}$ with the obvious meaning that $x = 1$ means taking the first alternative whereas $x = 0$ indicates the rejection of the first alternative and hence the selection of the second option.

Many practical decision processes can be represented by an appropriate combination of several binary decisions. This means that the overall decision problem consists of choosing one of two alternatives for a large number of binary decisions which may all influence each other. In the basic version of a *linear decision model* the outcome of the complete decision process is evaluated by a linear combination of the values associated with each of the binary decisions. In order to take pairwise interdependencies between decisions into account also a *quadratic function* can be used to represent the outcome of a decision process. The feasibility of a particular selection of alternatives may be very complicated to establish in practice because the binary decisions may influence or even contradict each other.

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

Formally speaking, the linear decision model is defined by n binary variables $x_j \in \{0,1\}$ which correspond to the selection in the jth binary decision and by *profit values* p_j which indicate the difference of value attained by choosing the first alternative, i.e. $x_j = 1$, instead of the second alternative ($x_j = 0$). Without loss of generality we can assume that after a suitable assignment of the two options to the two cases $x_j = 1$ and $x_j = 0$, we always have $p_j \geq 0$. The overall profit value associated with a particular choice for all n binary decisions is given by the sum of all values p_j for all decisions where the first alternative was selected.

In the following we will consider decision problems where the feasibility of a particular selection of alternatives can be evaluated by a linear combination of coefficients for each binary decision. In this model the feasibility of a selection of alternatives is determined by a *capacity restriction* in the following way. In every binary decision j the selection of the first alternative ($x_j = 1$) requires a *weight* or *resource* w_j whereas choosing the second alternative ($x_j = 0$) does not. A selection of alternatives is feasible if the sum of weights over all binary decisions does not exceed a given threshold capacity value c. This condition can be written as $\sum_{j=1}^{n} w_j x_j \leq c$. Considering this decision process as an optimization problem, where the overall profit should be as large as possible, yields the *knapsack problem* (KP), the core problem of this book.

This characteristic of the problem gives rise to the following interpretation of (KP) which is more colourful than the combination of binary decision problems. Consider a mountaineer who is packing his knapsack (or rucksack) for a mountain tour and has to decide which items he should take with him. He has a large number of objects available which may be useful on his tour. Each of these items numbered from 1 to n would give him a certain amount of comfort or benefit which is measured by a positive number p_j. Of course, the weight w_j of every object which the mountaineer puts into his knapsack increases the load he has to carry. For obvious reasons, he wants to limit the total weight of his knapsack and hence fixes the maximum load by the capacity value c.

In order to give a more intuitive presentation, we will use this knapsack interpretation and will usually refer to a "packing of items into a knapsack" rather than to the "combination of binary decisions" throughout this book. Also the terms "profit" and "weight" are based on this interpretation. Instead of making a number of binary decisions we will speak of the selection of a subset of items from the *item set* $N := \{1,\ldots,n\}$.

The knapsack problem (KP) can be formally defined as follows: We are given an *instance* of the knapsack problem with item set N, consisting of n *items* j with *profit* p_j and *weight* w_j, and the *capacity value* c. (Usually, all these values are taken from the positive integer numbers.) Then the objective is to select a subset of N such that the total profit of the selected items is maximized and the total weight does not exceed c.

Alternatively, a knapsack problem can be formulated as a solution of the following linear integer programming formulation:

$$\text{(KP)} \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{1.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \tag{1.2}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n. \tag{1.3}$$

We will denote the *optimal solution vector* by $x^* = (x_1^*,\ldots,x_n^*)$ and the *optimal solution value* by z^*. The set X^* denotes the *optimal solution set*, i.e. the set of items corresponding to the optimal solution vector.

Problem (KP) is the simplest non-trivial integer programming model with binary variables, only one single constraint and only positive coefficients. Nevertheless, adding the integrality condition (1.3) to the simple linear program (1.1)-(1.2) already puts (KP) into the class of "difficult" problems. The corresponding complexity issues will be addressed in Section 1.5.

The knapsack problem has been studied for centuries as it is the simplest prototype of a maximization problem. Already in 1897 Mathews [338] showed how several constraints may be aggregated into one single knapsack constraint. This is somehow a prototype of a reduction of a general integer program to (KP), thus proving that (KP) is at least as hard to solve as an integer program. It is however unclear how the name "Knapsack Problem" was invented. Dantzig is using the expression in his early work and thus the name could be a kind of folklore.

Considering the above characteristic of the mountaineer in the context of business instead of leisure leads to a second classical interpretation of (KP) as an investment problem. A wealthy individual or institutional investor has a certain amount of money c available which she wants to put into profitable business projects. As a basis for her decisions she compiles a long list of possible investments including for every investment the required amount w_j and the expected net return p_j over a fixed period. The aspect of risk is not explicitly taken into account here. Obviously, the combination of the binary decisions for every investment such that the overall return on investment is as large as possible can be formulated by (KP).

A third illustrating example of a real-world economic situation which is captured by (KP) is taken from airline cargo business. The dispatcher of a cargo airline has to decide which of the transportation requests posed by the customers he should fulfill, i.e. how to load a particular plane. His decision is based on a list of requests which contain the weight w_j of every package and the rate per weight unit charged for each request. Note that this rate is not fixed but depends on the particular long-term arrangements with every customer. Hence the profit p_j made by the company by accepting a request and by putting the corresponding package on the plane is

not directly proportional to the weight of the package. Naturally, every plane has a specified maximum capacity c which may not be exceeded by the total weight of the selected packages. This logistic problem is a direct analogon to the packing of the mountaineers knapsack.

While the previous examples all contain elements of "packing", one may also view the (KP) as a "cutting" problem. Assume that a sawmill has to cut a log into shorter pieces. The pieces must however be cut into some predefined standard-lengths w_j, where each length has an associated selling price p_j. In order to maximize the profit of the log, the sawmill can formulate the problem as a (KP) where the length of the log defines the capacity c.

An interesting example of the knapsack problem from academia which may appeal to teachers and students was reported by Feuerman and Weiss [144]. They describe a test procedure from a college in Norwalk, Connecticut, where the students may select a subset of the given question. To be more precise, the students receive n questions each with a certain "weight" indicating the number of points that can be scored for that question with a total of e.g. 125 points. However, after the exam all questions answered by the students are graded by the instructor who assigns points to each answer. Then a subset of questions is selected to determine the overall grade such that the maximum number of reachable points for this subset is below a certain threshold, e.g. 100. To give the students the best possible marks the subset should be chosen automatically such that the scored points are as large as possible. This task is clearly equivalent to solving a knapsack problem with an item j for the j-th question with w_j representing the reachable points and p_j the actually scored points. The capacity c gives the threshold for the limit of points of the selected questions.

However, as it is frequently the case with industrial applications, in practice several additional constraints, such as urgency and priority of requests, time windows for every request, packages with low weight but high volume etc., have to be fulfilled. This leads to various extensions and variations of the basic model (KP). Because this need for extension of the basic knapsack model arose in many practical optimization problems, some of the more general variants of (KP) have become standard problems of their own. We will introduce several of them in the following section and deal with many others in the later chapters of this book.

Beside these explicit occurrences of knapsack problems it should be noted that many solution methods of more complex problems employ the knapsack problem (sometimes iteratively) as a subproblem. Therefore, a comprehensive study of the knapsack problem carries many advantages for a wide range of mathematical models.

From a didactic and historic point of view it is worth mentioning that many techniques of combinatorial optimization and also of computer science were introduced in the context of, or in connection with knapsack problems. One of the first optimization problems to be considered in the development of $\mathcal{NP}$-hardness (see Section 1.5) was the subset sum problem (see Section 1.2). Other concepts such as

approximation schemes, reduction algorithms and dynamic programming were established in their beginning based on or illustrated by the knapsack problem.

Research in combinatorial optimization and operational research can be carried out either in a top-down or bottom-up fashion. In the top-down approach, researchers develop solution methods for the most difficult optimization problems like the *traveling salesman problem*, *quadratic assignment problem* or *scheduling problem*. If the developed methods work for these difficult problems, we may assume that they will also work for a large variety of other problems. The opposite approach is to develop new methods for the most simple model, like e.g. the *knapsack problem*, hoping that the techniques can be generalized to more complex models. Since both approaches are *method developing*, they justify a considerable research effort for solving a relatively simple problem.

Skiena [437] reports an analysis of a quarter of a million requests to the Stony Brook Algorithms Repository, to determine the relative level of interest among 75 algorithmic problems. In this analysis it turns out that codes for the knapsack problem are among the top-twenty of the most most requested algorithms. When comparing the interest to the number of actual knapsack implementations, Skiena concludes that knapsack algorithms are the third most needed implementations. Of course, research should not be driven by demand figures alone, but the analysis indicates that knapsack problems occur in many real-life applications and the solution of these problems is of vital interest both to the industry and administration.

1.2 Variants and Extensions of the Knapsack Problem

Let us consider again the previous problem of the cargo airline dispatcher. In a different setting the profit made by accepting a package may be directly proportional to its weight. In this case the optimal loading of the plane is achieved by filling as much weight as possible into the plane or equivalently setting $p_j = w_j$ in (KP). The resulting optimization problem is known as the *subset sum problem* (SSP) because we are looking for a *subset* of the values w_j with the *sum* being as close as possible to, but not exceeding the given target value c. It will be treated in Chapter 4.

$$
\begin{aligned}
\text{(SSP)} \qquad & \text{maximize} \quad \sum_{j=1}^{n} w_j x_j \\
& \text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \\
& \qquad\qquad x_j \in \{0,1\}, \quad j = 1,\ldots,n.
\end{aligned}
$$

In the original cargo problem described above it will frequently be the case that not all packages are different from each other. In particular, in practice there may be given a number b_j of identical copies of each item to be transported. If we either have to accept a request for all b_j packages or reject them all then we can generate an artificial request with weight $b_j w_j$ generating a profit of $b_j p_j$. If it is possible to select also only a subset of the b_j items from a request we can either represent each individual package by a binary variable or more efficiently represent the whole set of identical packages by an integer variable $x_j \geq 0$ indicating the number of packages of this type which are put into the plane. In this case the number of variables is equal to the number of different packages instead of the total number of packages. This may decrease the size of the model considerably if the numbers b_j are relatively large. Formally, constraint (1.3) in (KP) is replaced by

$$0 \leq x_j \leq b_j, \; x_j \text{ integer}, \quad j = 1, \ldots, n. \tag{1.4}$$

The resulting problem is called the *bounded knapsack problem* (BKP). Chapter 7 is devoted to (BKP). A special variant thereof is the *unbounded knapsack problem* (UKP) (see Chapter 8). It is also known as *integer knapsack problem* where instead of a fixed number b_j a very large or an infinite amount of identical copies of each item is given. In this case, constraint (1.3) in (KP) is simply replaced by

$$x_j \geq 0, \; x_j \text{ integer}, \quad j = 1, \ldots, n. \tag{1.5}$$

Moving in a different direction, we consider again the above cargo problem and now take into account not only the weight constraint but also the limited space available to transport the packages. For practical purposes only the volume of the packages is considered and not their different shapes.

Denoting the weight of every item by w_{1j} and its volume by w_{2j} and introducing the weight capacity of the plane as c_1 and the upper bound on the volume as c_2 we can formulate the extended cargo problem by replacing constraint (1.2) in (KP) by the two inequalities

$$\sum_{j=1}^{n} w_{1j} x_j \leq c_1,$$

$$\sum_{j=1}^{n} w_{2j} x_j \leq c_2.$$

The obvious generalization of this approach, where d instead of two inequalities are introduced, yields the *d-dimensional knapsack problem* or *multidimensional knapsack problem* (see Chapter 9) formally defined by

$$\text{(d-KP)} \qquad \text{maximize} \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i = 1,\ldots,d,$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

Another interesting variant of the cargo problem arises from the original version described above if we consider a very busy flight route, e.g. Frankfurt – New York, which is flown by several planes every day. In this case the dispatcher has to decide on the loading of a number of planes in parallel, i.e. it has to be decided whether to accept a particular transportation request and in the positive case on which plane to put the corresponding package. The concepts of profit, weight and capacity remain unchanged. This can be formulated by introducing a binary decision variable for every combination of a package with a plane. If there are n items on the list of transportation requests and m planes available on this route we use nm binary variables x_{ij} for $i = 1,\ldots,m$ and $j = 1,\ldots,n$ with

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is put into plane } i, \\ 0 & \text{otherwise.} \end{cases} \tag{1.6}$$

The mathematical programming formulation of this *multiple knapsack problem* (MKP) is given by

$$\text{(MKP)} \qquad \text{maximize} \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \tag{1.7}$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i = 1,\ldots,m, \tag{1.8}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad j = 1,\ldots,n, \tag{1.9}$$

$$x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m,\ j = 1,\ldots,n.$$

Constraint (1.9) guarantees that every item is put at most into one plane. If the capacities of the planes are identical we can easily simplify the above model by introducing a capacity c for all planes and by replacing constraints (1.8) by

$$\sum_{j=1}^{n} w_j x_{ij} \leq c, \quad i = 1,\ldots,m. \tag{1.10}$$

The latter model is frequently referred to as the *multiple knapsack problem with identical capacities* (MKP-I). Note that for both versions of multiple knapsacks also

the subset sum variants with $p_j = w_j$ were investigated. In this case the objective (1.7) is replaced by

$$\text{maximize} \quad \sum_{i=1}^{m} \sum_{j=1}^{n} w_j x_{ij}. \tag{1.11}$$

In analogy to the knapsack problem this defines the *multiple subset sum problem* (MSSP) with arbitrary capacities given by constraint (1.8) or the *multiple subset sum problem* (MSSP-I) with identical capacities and constraint (1.10). (MKP) and its variants will be investigated in Chapter 10.

A quite different variant of the cargo problem appears if a single plane is used to transport the equipment of an expedition to a deserted place. The expedition needs a number of tools like a rubber dinghy, a vehicle, special instruments etc. Each tool i however exists in a number of variants where the j-th variant has weight w_{ij} and utility value p_{ij}. As the plane can carry only a limited capacity c the objective is to select one variant of each tool such that the overall utility value is maximized without exceeding the capacity constraint.

This problem may be expressed as the following *multiple-choice knapsack problem* (MCKP). Assume that N_i is the set of different variants of tool i. Using the decision variables x_{ij} to denote whether variant j was chosen from the set N_i, the following model appears:

$$\begin{aligned}
\text{(MCKP)} \qquad \text{maximize} \quad & \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq c, \\
& \sum_{j \in N_i} x_{ij} = 1, \quad i = 1, \ldots, m, \\
& x_{ij} \in \{0,1\}, \quad i = 1, \ldots, m,\ j \in N_i.
\end{aligned} \tag{1.12}$$

Constraint (1.12) ensures that exactly one tool is chosen from each class. Chapter 11 is devoted to (MCKP) and its variants.

Another variant appears if an item j has a corresponding profit p_{jj} and an additional profit p_{ij} is redeemed only if item j is transported together with another item i which may reflect how well the given items fit together. This problem is expressed as the *quadratic knapsack problem* (QKP) (see Chapter 12) which is formally defined as follows:

$$\text{(QKP)} \qquad \text{maximize} \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}x_ix_j$$

$$\text{subject to} \sum_{j=1}^{n} w_jx_j \leq c, \tag{1.13}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

Besides the classical variants of (KP) which have been introduced in this section there are a lot of other knapsack-type which will be presented in Chapter 13.

1.3 Single-Capacity Versus All-Capacities Problem

A natural extension of the knapsack problem is to solve the problem not only for one given capacity c, but for all capacities c up to a given upper limit $c_{\max}$. We will denote this problem as the *all-capacities knapsack problem*. This variant of the problem has been overlooked in the literature, but it is equally relevant as solving the *all-pairs shortest path problem* as opposed to the *single-source shortest path problem*. In several planning problems, the exact capacity is not known in advance but may be negotiated on basis of the proposed solutions.

To continue our previous example with the cargo airline, we may notice that the capacity of an airplane depends on the amount of fuel on board. Carrying more cargo means that more fuel is needed, but the fuel also counts in the overall cargo weight. Since the fuel versus cargo function is not linear we may need to solve the all-capacities knapsack problem for all capacities up to a given upper limit on the cargo weight. For each capacity we then calculate the fuel demands and subtract the corresponding fuel costs from the profit. The optimal solution is found as the maximum of the final profits.

A natural method for solving the all-capacities knapsack problem is simply to solve (KP) for each capacity $0,\ldots,c_{\max}$. However, special algorithms may take benefit of the fact that the problem is solved for several capacities, such that the overall running time of solving the all-capacities knapsack problem is smaller than solving the c individual problems. In Section 2.3 we will show that specific algorithms based on *dynamic programming* are able to solve the all-capacities knapsack problem using the same computational effort as solving a single capacity knapsack problem.

1.4 Assumptions on the Input Data

To avoid trivial situations and tedious considerations of pointless sub-cases we will impose a number of assumptions on the input data of the knapsack problem which

will be valid throughout all chapters. However, as described below this will not result in a loss of generality. The analogous assumptions apply also to most of the variants of (KP). If necessary, details will be given in the corresponding chapters.

First of all, only problems with at least two items will be considered, i.e. we assume $n \geq 2$. Obviously, the case $n = 1$ represents a single binary decision which can be made by simply evaluating the two alternatives.

Two other straightforward assumptions concern the item weights. Naturally, we can never pack an item into the knapsack if its weight exceeds the capacity. Hence, we can assume

$$w_j \leq c, \quad j = 1,\ldots,n, \tag{1.14}$$

because otherwise we would have to set to 0 any binary variable corresponding to an item violating (1.14). If on the other hand all items together fit into the knapsack, the problem is trivially solved by packing them all. Therefore, we assume

$$\sum_{j=1}^{n} w_j > c. \tag{1.15}$$

Otherwise, we would set $x_j = 1$ for $j = 1,\ldots,n$.

Without loss of generality we may assume that all profits and weights are positive

$$p_j > 0, \quad w_j > 0, \quad j = 1,\ldots,n. \tag{1.16}$$

If this is not the case we may transform the instance to satisfy (1.16) as follows. Depending on the sign of the coefficients of a given item one of the following actions take place (excluding the case of a trivial item with $p_j = w_j = 0$, where x_j can be chosen arbitrarily).

1. $p_j \geq 0$ and $w_j \leq 0$: Set $x_j = 1$. Since item j does not increase the left hand side of the capacity constraint (1.2) but possibly increases the objective function it is always better to pack item j rather than leaving it unpacked.
2. $p_j \leq 0$ and $w_j \geq 0$: Set $x_j = 0$. Packing item j into the knapsack neither increases the objective function nor increases the available capacity. Hence, it will always be better not to pack this item.
3. $p_j < 0$ and $w_j < 0$: This is a more interesting case which can be seen as a possibility to sacrifice some profit in order to increase the available capacity thus "making room" for the weight of a "more attractive" item. It can be handled by the following construction. Assume that items belonging to the two cases above have been dealt with and hence are eliminated from N. Now the set of items $N = \{1,\ldots,n\}$ can be partitioned into $N_+ := \{j \mid p_j > 0, w_j > 0\}$ and $N_- := \{j \mid p_j < 0, w_j < 0\}$. For all $j \in N_-$ define $\tilde{p}_j := -p_j$ and $\tilde{w}_j := -w_j$. Furthermore, let us introduce the binary variable $y_j \in \{0,1\}$ as the complement of x_j with the meaning

$$y_j = \begin{cases} 1 & \text{if item } j \text{ is } \textit{not} \text{ packed into the knapsack,} \\ 0 & \text{it item } j \text{ is packed into the knapsack.} \end{cases} \tag{1.17}$$

Now we can formulate the following standard knapsack problem with positive coefficients. It can be interpreted as putting all items from N_- into an enlarged knapsack before the optimization and then getting a positive profit $\tilde{p}_j$ consuming a positive capacity $\tilde{w}_j$ if $y_j = 1$, i.e. if the item j is unpacked again from the knapsack.

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in N_+} p_j x_j + \sum_{j \in N_-} \tilde{p}_j y_j + \sum_{j \in N_-} p_j \\ \text{subject to} \quad & \sum_{j \in N_+} w_j x_j + \sum_{j \in N_-} \tilde{w}_j y_j \leq c - \sum_{j \in N_-} w_j, \\ & x_j \in \{0,1\},\ j \in N_+, \quad y_j \in \{0,1\},\ j \in N_-. \end{aligned}$$

A rather subtle point is the question of rational coefficients. Indeed, most textbooks get rid of this case, where some or all input values are noninteger, by the trivial statement that multiplying with a suitable factor, e.g. with the smallest common multiple of the denominators, if the values are given as fractions or by a suitable power of 10, transforms the data into integers. Clearly, this may transform even a problem of moderate size into a rather unpleasant problem with huge coefficients. As we will see later, the magnitude of the input values has a considerable negative effect on the performance of many algorithms (see Section 5.5). However, some methods, especially branch-and-bound methods (see Section 2.4), can be adapted relatively easily to accommodate rational input values. Also, approximation algorithms are normally unaffected by the coefficient values since some kind of scaling of the coefficients is made in any case. All dynamic programming algorithms also work well as long as at least either the profits p_j or the weights w_j are integers.

1.5 Performance of Algorithms

The main goal of studying knapsack problems is the development of solution methods, i.e. algorithms, which compute an optimal or an approximate solution for every given problem instance. Clearly, not all algorithms which compute an optimal solution are equivalent in their performance. Moreover, it seems natural that an algorithm which computes only approximate but not necessarily optimal solutions should make up for this drawback in some sense by a better computational behaviour.

In this context performance is generally defined by the running time and the amount of computer memory, i.e. space, required to solve the given problem. Another important aspect would be the level of difficulty of an algorithm because "easy" methods

will be clearly preferred to very complicated algorithms which are more costly to implement. However, this aspect is very difficult to quantify since the implementation costs are dependent on many factors such as experience or computational environment. Anyway, it should be clear from the description of the algorithms whether they are straightforward or rather exhausting to implement.

Obviously, it is of crucial interest to have some measure in order to distinguish "faster" and "slower" algorithms. Basically, there are three ways to introduce criteria for characterizing algorithms. The first approach would be to implement the algorithm and test it for different problem instances. Naturally, any comparison of computer programs has to be made with utmost diligence to avoid confounding factors such as different hardware, different software environment and different quality of implementation. A major difficulty is also the selection of appropriate test instances. We will report computational results in separate sections throughout the book. We refer especially to Sections 4.4, 4.7, 5.5, 7.3, 11.8, 12.8 and 12.9.

A second way would be the investigation of the *average running time* of an algorithm. However, it is by no means clear what kind of "average" we are looking for. Finding a suitable probabilistic model which reflects the occurrence of real-world instances is quite a demanding task on its own. Moreover, the technical difficulties of dealing with such probabilistic models are overwhelming for most of the more complicated algorithms and only very simple probabilistic models can be analyzed. Another major drawback of the method is the fact that most results are relevant only if the size of the problem, i.e. the number of items, is sufficiently large, or if a sufficiently large number of samples is performed. Some results in this direction will be described in Chapter 14.

The third and most common method to measure the performance of an algorithm is the *worst-case analysis* of its running time. This means that we give an upper bound on the number of basic arithmetic operations required to solve *any* instance of a given size. This upper bound should be a function of the size of the problem instance, which depends on the number of input values and also on their magnitude. Consider that we may have to perform a certain set of operations for every item or e.g. for every integer value from one up to the maximum capacity c. For most running time bounds in this book it will be sufficient to consider as parameters for the size of a problem instance the number of items n, the capacity c and the largest profit or weight value $p_{\max} := \max\{p_j \mid j = 1,\ldots,n\}$ or $w_{\max} := \max\{w_j \mid j = 1,\ldots,n\}$. The size of a problem instance is discussed in the more rigorous sense of theoretical computer science in Appendix A.

Counting the exact number of the necessary arithmetic operations of an algorithm is almost impossible and also depends on implementational details. In order to compare different algorithms we are mainly interested in an *asymptotic upper bound* for the running time to illustrate the order of magnitude of the increase in running time, i.e. we want to know the increase in running time if for example the number of items is doubled. Constant factors or constant additive terms are ignored which allows for greater generality since implementational "tricks" and machine specific

features play no role in this representation. An analogous procedure applies to the required amount of computer memory.

These asymptotic running time bounds are generally described in the so-called O-notation. Informally, we can say that every polynomial in n with largest exponent k is in $O(n^k)$. This means that we neglect all terms with exponents smaller than k and also the constant coefficient of n^k. More formally, for a given function $f : \mathbb{R} \to \mathbb{R}$ the expression $O(f)$ denotes the family of all functions

$$O(f) := \{g(x) \mid \exists c, x_0 > 0 \text{ s.t. } 0 \leq g(x) \leq cf(x) \; \forall x \geq x_0\}.$$

Let us consider e.g. the function $f(x) := x^2$. Then all of the following functions are in fact included in $O(f)$:

$$10x,\ x\log x,\ 0.1x^2,\ 1000x^2,\ x^{1.5}$$

However, $x^{2+\varepsilon}$ is not in $O(f)$ for any $\varepsilon > 0$.

More restrictive is the so-called Θ-notation which asymptotically bounds a function from below and above. Formally, we denote by $\Theta(f)$ the family of all functions

$$\Theta(f) := \{g(x) \mid \exists c_1, c_2, x_0 > 0 \text{ s.t. } 0 \leq c_1 f(x) \leq g(x) \leq c_2 f(x) \; \forall x \geq x_0\}.$$

Throughout this text we will briefly write that the running time of an algorithm e.g. is $O(n^2)$, meaning the there is an asymptotic upper bound on the running time which is in the family $O(n^2)$. The same notation is also used to describe the space requirements of an algorithm. For more information and examples of the O-notation the reader may consult any textbook on algorithms, e.g. the book by Cormen et. al. [92, Section 3.1]. We would like to note that although a very complicated $O(n\log n)$ algorithm is "faster" than a simple and straightforward $O(n^2)$ method, in practice this superiority may become relevant only for large problem instances, as some huge constants may be hidden in the Big-Oh notation.

Depending on their asymptotic running time bounds algorithms can be partitioned into three classes. Here, we will give a rather informal illustration of these classes whereas a rigorous treatment can be found in the Appendix A.

The most efficient algorithms are those where the running time is bounded by a *polynomial* in n, e.g. $O(n)$, $O(n\log n)$, $O(n^3)$ or $O(n^k)$ for a constant k. These are also called *polynomial time algorithms*. The so-called *pseudopolynomial algorithms*, where the running time is bounded asymptotically by a polynomial both in n and in one (or several) of the input values, e.g. $O(nc)$ or $O(n^2 p_{\max})$, are less "attractive" because even a simple problem with a small number of items may have quite large coefficients and hence a very long running time. The third and most unpleasant class of problems are the *non-polynomial algorithms*, i.e. those algorithms where the running time cannot be bounded by a polynomial but e.g. only by an exponential function in n, e.g. $O(2^n)$ or $O(3^n)$. Their unpleasantness can be illustrated by

comparing e.g. $O(n^3)$ to $O(2^n)$. If n is increased to $2n$ the first expression increases by a factor of 8 whereas the second expression is squared!

For the knapsack problem and many of its generalizations pseudopolynomial algorithms are known. Clearly, we would prefer to have even polynomial time algorithms for these problems. However, there is very strong theoretical evidence that for the knapsack problem and hence also for its generalizations no polynomial time algorithm exists for computing its optimal solution. In fact, all these problems belong to a class of so-called *$\mathcal{NP}$-hard optimization problems*. It is widely believed that there does not exist any polynomial time algorithm to solve an $\mathcal{NP}$-hard problem to optimality because all $\mathcal{NP}$-hard problems are equivalent in the sense that if any $\mathcal{NP}$-hard problem would be found out to be solvable in polynomial time then this property would apply to *all* $\mathcal{NP}$-hard problems. A more formal treatment of these issues can be found in the Appendix A.

2. Basic Algorithmic Concepts

In this chapter we introduce some algorithmic ideas which will be used in more elaborate ways in various chapters for different problems. The aim of presenting these ideas in a simple form is twofold. On one hand we want to provide the reader, who is a novice in the area of knapsack problems or combinatorial and integer programming in general, with a basic introduction such that no other reference is needed to work successfully with the more advanced algorithms in the remainder of this book. On the other hand, we would like to establish a common conceptual cornerstone from which the other chapters can develop the more refined variants of these basic ideas as they are required for each of the individual problems covered there. In this way we can avoid repetitions without affecting the self-contained character of the remaining chapters.

We will also introduce some further notation. Let $S \subseteq N$ be a subset of the item set N. Then $p(S)$ and $w(S)$ denote the total profit and the total weight of the items in set S, respectively. Recall from Section 1.5 that $p_{\max} = \max\{p_j \mid j = 1,\dots,n\}$ and $w_{\max} = \max\{w_j \mid j = 1,\dots,n\}$ denote the largest profit and the largest weight value, respectively. Analogously, $p_{\min} := \min\{p_j \mid j = 1,\dots,n\}$ and $w_{\min} := \min\{w_j \mid j = 1,\dots,n\}$ denote the smallest profit and weight values.

2.1 The Greedy Algorithm

If a non-expert were trying to find a good solution for the knapsack problem (KP), i.e. a profitable packing of items into the knapsack, an intuitive approach would be to consider the *profit to weight ratio* e_j of each item which is also called the *efficiency* of an item with

$$e_j := \frac{p_j}{w_j}, \tag{2.1}$$

and try to put the items with highest efficiency into the knapsack. Clearly, these items generate the highest profit while consuming the lowest amount of capacity.

Therefore, in this section we will assume the items to be sorted by their efficiency in decreasing order such that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}. \tag{2.2}$$

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

Note that this ordering will not be presupposed throughout this book but only where explicitly stated. This is done to distinguish precisely which algorithms require property (2.2) and which not and to include the necessary effort for sorting in all running time estimations.

The idea of the *greedy algorithm* with solution value z^G is to start with an empty knapsack and simply go through the items in this decreasing order of efficiencies adding every item under consideration into the knapsack if the capacity constraint (1.2) is not violated thereby.

An explicit description of this algorithm Greedy is given in Figure 2.1.

Algorithm Greedy:

$\overline{w} := 0$ $\quad$ *$\overline{w}$ is the total weight of the currently packed items*
$z^G := 0$ $\quad$ *z^G is the profit of the current solution*
for $j := 1$ **to** n **do**
  if $\overline{w} + w_j \leq c$ **then**
    $x_j := 1$ $\quad$ *put item j into the knapsack*
    $\overline{w} := \overline{w} + w_j$
    $z^G := z^G + p_j$
  else $x_j := 0$

Fig. 2.1. Algorithm Greedy adding the items in decreasing order of efficiency.

A slight variation of Greedy is algorithm Greedy-Split which stops as soon as algorithm Greedy failed for the first time to put an item into the knapsack. After sorting the items according to (2.2) in $O(n \log n)$ time the running time both of Greedy and Greedy-Split is $O(n)$ since every item is considered at most once. It will be shown in Section 3.1 that Greedy-Split can be implemented to run even in $O(n)$ without presorting the items. Besides storing the input data and the solution no additional space is required.The "quality" of the solution computed by Greedy will be discussed in Section 2.5 where it will be shown that the Greedy solution may be arbitrarily bad compared to the optimal solution. However, it will also be shown that a small extension of Greedy yields an algorithm which always computes a packing of the knapsack with a total profit at least half as large as the optimal solution value (see Theorem 2.5.4).

Example: We consider an instance of (KP) with capacity $c = 9$ and $n = 7$ items with the following profit and weight values:

j	1	2	3	4	5	6	7
p_j	6	5	8	9	6	7	3
w_j	2	3	6	7	5	9	4

The items are already sorted by their efficiency in decreasing order. Algorithm Greedy puts items 1,2 and 7 into the knapsack yielding a profit $z^G = 14$, whereas Greedy-Split stops after assigning items 1 and 2 with total profit 11. The optimal solution $X^* = \{1,4\}$ has a solution value of 15. □

2.2 Linear Programming Relaxation

A standard approach for any integer program to get more insight into the structure of the problem and also to get a first estimate of the solution value is the omission of the integrality condition. This *linear programming relaxation* (LKP) or *LP-relaxation* of the original problem is derived by optimizing over all (usually nonnegative) real values instead of only the integer values. In the case of (KP) this means that (1.3) is replaced by $0 \leq x_j \leq 1$ for $j = 1,\ldots,n$ thus getting the problem

$$\begin{aligned} \text{(LKP)} \qquad & \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \\ & \text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \\ & \qquad\qquad 0 \leq x_j \leq 1, \quad j = 1,\ldots,n. \end{aligned} \tag{2.3}$$

Naturally, for a maximization problem the optimal solution value z^{LP} of the relaxed problem is at least as large as the original solution value z^* because the set of feasible solutions for (KP) is a subset of the feasible solutions for the relaxed problem.

The solution of (LKP) can be computed in a very simple way as (LKP) possesses the *greedy choice property*, i.e. a global optimum can be obtained by making a series of greedy (locally optimal) choices.

The greedy choice for (LKP) is to pack the items in decreasing order of their efficiencies, thus getting the highest profit per each weight unit in every step. Hence, we will assume in this section that the items are sorted and (2.2) holds. We may use the above algorithm Greedy to solve (LKP) with minor modifications. If adding an item to the knapsack would cause an overflow of the capacity for the first time, let us say in iteration s, i.e. if

$$\sum_{j=1}^{s-1} w_j \leq c \quad \text{and} \quad \sum_{j=1}^{s} w_j > c, \tag{2.4}$$

then the execution of Greedy is stopped and the residual capacity $c - \sum_{j=1}^{s-1} w_j$ is filled by an appropriate fractional part of item s.

Item s is frequently referred to as the *split item*. Other authors use the term *break item* or *critical item*. The solution vector $\hat{x}$ with $\hat{x}_j = 1$ for $j = 1,\ldots,s-1$ and $\hat{x}_j = 0$

for $j = s, \ldots, n$ will be denoted as the *split solution*. Analogously, let $\hat{p}$ and $\hat{w}$ be the profit and weight of the split solution. Note that the split solution is identical to the solution produced by Greedy-Split.

The following theorem defines the optimal solution to (LKP) and thus proves that the greedy choice property holds for (LKP).

Theorem 2.2.1 *An optimal solution vector* $x^{LP} = (x_1^{LP}, \ldots, x_n^{LP})$ *of (LKP) is defined as follows:*

$$\begin{aligned} x_j^{LP} &:= 1, && j = 1, \ldots, s-1, \\ x_s^{LP} &:= \tfrac{1}{w_s}\left(c - \textstyle\sum_{j=1}^{s-1} w_j\right), \\ x_j^{LP} &:= 0, && j = s+1, \ldots, n. \end{aligned}$$

The corresponding solution value is

$$z^{LP} = \sum_{j=1}^{s-1} p_j + \left(c - \sum_{j=1}^{s-1} w_j\right) \frac{p_s}{w_s}. \tag{2.5}$$

Proof. In the proof we will assume that items with the same efficiency $e_j = p_j/w_j$ have been merged into a single item. As we are allowed to choose any fraction of an item this does not change the problem but we avoid discussing symmetric solutions. The sorting (2.2) will now satisfy that

$$\frac{p_1}{w_1} > \frac{p_2}{w_2} > \ldots > \frac{p_n}{w_n}. \tag{2.6}$$

Assume that $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$ is an optimal solution vector for (LKP) different from x^{LP}. Clearly, every optimal solution of (LKP) will have a total weight equal to c. If $\bar{x}$ and x^{LP} differ only in item s then $\bar{x}$ does not consume all the available capacity and cannot be optimal. Thus there must exist at least one variable $k > s$ such that $\bar{x}_k > 0$. Hence also a variable $i < s$ must exist such that $\bar{x}_i < 1$. Let

$$d := \min\{w_k \bar{x}_k, w_i(1 - \bar{x}_i)\} > 0.$$

Construct a new solution vector $x' := (x'_1, \ldots, x'_n)$ which is identical to $\bar{x}$ except that $x'_k = \bar{x}_k - d/w_k$ and that $x'_i = \bar{x}_i + d/w_i$. x' is well-defined since $x'_k \geq 0$ due to the definition of d. The weight sum of x' is

$$\sum_{j=1}^{n} w_j x'_j = \sum_{j=1}^{n} w_j \bar{x}_j + \frac{w_i d}{w_i} - \frac{w_k d}{w_k} = c,$$

thus x' is a feasible solution. The profit sum is however

$$\sum_{j=1}^{n} p_j x'_j = \sum_{j=1}^{n} p_j \bar{x}_j + d\left(\frac{p_i}{w_i} - \frac{p_k}{w_k}\right) > \sum_{j=1}^{n} p_j \bar{x}_j,$$

as $p_i/w_i > p_k/w_k$, contradicting the fact that $\bar{x}$ is an optimal solution. □

The value z^{LP} is an upper bound on the optimal solution. A tighter upper bound U_{LP} for z^* can be obtained by rounding down z^{LP} to the next integer, i.e. $U_{\mathrm{LP}} := \lfloor z^{LP} \rfloor$, since all data are integers. Then we get the following bounds on z^{LP} as trivial consequence of Theorem 2.2.1.

Corollary 2.2.2 $\hat{p} \leq z^* \leq U_{LP} \leq z^{LP} \leq \sum_{j=1}^{s} p_j \leq \hat{p} + p_s \leq z^G + p_s$

Another straightforward consequence of these considerations is the following useful fact.

Corollary 2.2.3 $z^* - z^G \leq z^* - \hat{p} \leq p_{\max}$

Example: (cont.) Consider the example of Section 2.1. Item 3 is the split item. We have $\hat{p} = 11$, $z^{LP} = 16\frac{1}{3}$ and $U_{\mathrm{LP}} = 16$. □

Computing the optimal solution vector x^{LP} is straightforward and can be done trivially in $O(n)$ time after the items were sorted according to (2.2) which requires $O(n \log n)$ time. However, even if the items are not sorted there exists a more advanced procedure which computes x^{LP} in $O(n)$ time. It will be presented in Section 3.1.

Although for many practical integer programming instances the difference between the integer optimal solution and the solution of the linear programming relaxation is quite small, the solution of (LKP) may be almost twice as large as the (KP) solution. Indeed, consider the following instance of (KP):

Let $n = 2$ and $c = 2M$. We are given two identical items with $w_j = M + 1$ and $p_j = 1$ for $j = 1, 2$. Obviously, the optimal solution can pack only one item into the knapsack generating an optimal solution value of 1, whereas an optimal solution of (LKP) is attained according to Theorem 2.2.1 by setting $x_1 = 1$ and $x_2 = \frac{M-1}{M+1}$ yielding the solution value $\frac{2M}{M+1}$ which is arbitrarily close to 2 for M chosen sufficiently large.

However, the (LKP) solution denoted by z^{LP} can never be more than twice as large as z^*, the optimal solution value of (KP), because the latter is on the one hand at least as large as $\hat{p} = \sum_{j=1}^{s-1} p_j$ and on the other hand at least as large as p_s. We conclude

Lemma 2.2.4 $z^{LP} \leq 2z^*$ *and there are instances such that* $z^{LP} \geq 2z^* - \varepsilon$ *for every* $\varepsilon > 0$.

2.3 Dynamic Programming

In the previous two subsections we tried to deal with the knapsack problem in an intuitive and straightforward way. However, both "natural" approaches turned out to deliver solutions respectively upper bounds which may be far away from the optimal solution. Since there is no obvious strategy of "guessing" or constructing the optimal solution of (KP) in one way or other from scratch, we might try to start by solving only a small subproblem of (KP) and then extend this solution iteratively until the complete problem is solved. In particular, it may be useful not to deal with all n items at once but to add items iteratively to the problem and its solution. This basic idea leads to the use of the concept of *dynamic programming*.

The technique of dynamic programming is a general approach which appears as a useful tool in many areas of operations research. Basically, it can be applied whenever an optimal solution consists of a combination of optimal solutions to subproblems. The classical book by Bellman [32] gives an extensive introduction into the field and can still be seen as a most useful general reference.

Considering an optimal solution of the knapsack problem it is obvious that by removing any item r from the optimal knapsack packing, the remaining solution set must be an optimal solution to the subproblem defined by capacity $c - w_r$ and item set $N \setminus \{r\}$. Any other choice will risk to diminish the optimal solution value. Hence, (KP) has the property of an *optimal substructure* as described e.g. in Cormen et al. [92, Section 15.3].

This basic property is the foundation for the classical application of dynamic programming to the knapsack problem which can be described as follows: Let us assume that the optimal solution of the knapsack problem was already computed for a subset of the items and all capacities up to c, i.e. for the all-capacities knapsack problem (see Section 1.3). Then we add one item to this subset and check whether the optimal solution needs to be changed for the enlarged subset. This check can be done very easily by using the solutions of the knapsack problems with smaller capacity as described below. To preserve this advantage we have to compute the possible change of the optimal solutions again for all capacities. This procedure of adding an item is iterated until finally all items were considered and hence the overall optimal solution is found.

More formally, we introduce the following subproblem of (KP) consisting of item set $\{1,\ldots,j\}$ and a knapsack capacity $d \leq c$. For $j = 0,\ldots,n$ and $d = 0,\ldots,c$ let $(\mathrm{KP}_j(d))$ be defined as

$$
\begin{aligned}
(\mathrm{KP}_j(d)) \qquad & \text{maximize} \quad \sum_{\ell=1}^{j} p_\ell x_\ell \\
& \text{subject to} \quad \sum_{\ell=1}^{j} w_\ell x_\ell \leq d,
\end{aligned}
\tag{2.7}
$$

$$x_\ell \in \{0,1\}, \quad \ell = 1,\ldots,j,$$

with optimal solution value $z_j(d)$. For convenience, the optimal subset of packed items for $\mathrm{KP}_j(d)$ is represented by a set $X_j(d)$.

If $z_{j-1}(d)$ is known for all capacity values $d = 0,\ldots,c$, then we can consider an additional item j and compute the corresponding solutions $z_j(d)$ by the following recursive formula

$$z_j(d) = \begin{cases} z_{j-1}(d) & \text{if } d < w_j, \\ \max\{z_{j-1}(d), z_{j-1}(d-w_j) + p_j\} & \text{if } d \geq w_j. \end{cases} \tag{2.8}$$

The recursion (2.8) is also known as the *Bellman recursion* [32]. The case $d < w_j$ means that we consider a knapsack which is too small to contain item j at all. Hence, item j does not change the optimal solution value $z_{j-1}(d)$. If item j does fit into the knapsack, there are two possible choices: Either item j is not packed into the knapsack and the previous solution $z_{j-1}(d)$ remains unchanged or item j is added into the knapsack which contributes p_j to the solution value but decreases the capacity remaining for items from $\{1,\ldots,j-1\}$ to $d - w_j$. Obviously, this remaining capacity should be filled with as much profit as possible. The best possible solution value for this reduced capacity is given by $z_{j-1}(d-w_j)$. Taking the maximum of these two choices yields the optimal solution for $z_j(d)$. The contents of the sets $X_j(d)$ follow immediately from this consideration. Whenever $z_j(d)$ is evaluated as $z_{j-1}(d-w_j)+p_j$ we have $X_j(d) := X_{j-1}(d-w_j) \cup \{j\}$. In all other cases we simply have $X_j(d) := X_{j-1}(d)$.

Initializing $z_0(d) := 0$ for $d = 0,\ldots,c$ and computing the values $z_j(d)$ by recursion (2.8) from $j = 1$ up to $j = n$ finally yields the overall optimal solution value of (KP) as $z_n(c)$. Since we consider the solution values as function of a capacity value, this version is also called *dynamic programming by weights*. Note that in this way we solve not only the given single-capacity knapsack problem but also the all-capacities knapsack problem. The resulting algorithm DP-1 is described in Figure 2.2.

Example: (cont.) Consider again the example of Section 2.1. Figure 2.3 shows the optimal solution values $z_j(d)$ for problem $(KP_j(d))$ with items $1,\ldots,j$ and capacity d computed by algorithm DP-1. The computation is done columnwise from left to right and inside a column from top to bottom. Row 0 and and column 0 are initialized with entries equal to 0. The values in boxes correspond to the profits of the optimal solution set. □

The running time of DP-1 is dominated by the n iterations of the second for–loop each of which contains at most $c+1$ iterations where a new solution of a subproblem is computed. This yields an overall running time of $O(nc)$. Hence, we have presented a pseudopolynomial algorithm for (KP). The necessary space requirement of this implementation would also be $O(nc)$. It must pointed out that DP-1 only computes

Algorithm DP-1:

for $d := 0$ to c do
  $z_0(d) := 0$ *initialization*
for $j := 1$ to n do
  for $d := 0$ to $w_j - 1$ do *item j is too large to be packed*
    $z_j(d) := z_{j-1}(d)$
  for $d := w_j$ to c do *item j may be packed*
    if $z_{j-1}(d - w_j) + p_j > z_{j-1}(d)$ then
      $z_j(d) := z_{j-1}(d - w_j) + p_j$
    else $z_j(d) := z_{j-1}(d)$
$z^* := z_n(c)$

Fig. 2.2. Basic dynamic programming algorithm DP-1 for (KP).

$d \backslash j$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	6	6	6	6	6	6	6
3	0	6	6	6	6	6	6	6
4	0	6	6	6	6	6	6	6
5	0	6	11	11	11	11	11	11
6	0	6	11	11	11	11	11	11
7	0	6	11	11	11	12	12	12
8	0	6	11	14	14	14	14	14
9	0	6	11	14	15	15	15	15

Fig. 2.3. The optimal solution values $z_j(d)$ computed by DP-1 for the example.

the optimal solution value z^* but does not explicitly return the corresponding optimal solution set X^*.

In a straightforward algorithm every subset $X_j(d)$ is represented as an array of length n, where entry ℓ is 1 or 0 indicating whether item ℓ is included in the optimal packing or not. Storing these sets $X_j(d)$ for every subproblem and updating them together with $z_j(d)$ would increase both the running time and the space requirements by a factor of n. This would yield an overall time and space complexity of $O(n^2c)$ for computing z^* and X^*.

Instead of using a separate storage position or computer word for every item we can encode the sets $X_j(d)$ by bit strings, i.e. every single bit of a computer word is used separately to indicate whether an item is included in the set or not. This would decrease running time and space by a factor equal to the number of bits in one computer word. However, in terms of complexity as given by the O-notation this constant factor is hardly relevant.

We can improve upon this straightforward set representation in the following way. Obviously, the item sets $X_j(d)$ computed in iteration j will differ from an item set

generated in the preceding iteration $j-1$ by at most one item, namely the possibly added item j. Hence, it is a waste of space to keep the full arrays $X_j(d)$ in every iteration. Instead, it is sufficient to store for every weight d in iteration j only the information whether item j was added to the knapsack or not. This can be done easily by keeping a pointer $A_j(d) \in \{0,1\}$ with the meaning

$$A_j(d) := \begin{cases} 1 & \text{if } z_j(d) = z_{j-1}(d-w_j) + p_j, \\ 0 & \text{if } z_j(d) = z_{j-1}(d), \end{cases} \tag{2.9}$$

for $j = 1,\ldots,n$ and $d = 1,\ldots,c$. It is easy to see how X^* can be reconstructed by going through the pointers. If $A_n(c) = 1$ then item n is added to X^* and we proceed with checking $A_{n-1}(c-w_n)$. If $A_n(c) = 0$ then item n is not part of the optimal solution set and we proceed with $A_{n-1}(c)$. The resulting version of DP-1 computes both z^* and X^* in $O(nc)$ time and space.

Looking at the values of $A_j(d)$ it can be observed that it is not really necessary to store this table explicitly. Note that every entry of $A_j(d)$, which is needed during the reconstruction of X^*, can be directly computed in constant time using (2.9) since all entries of $z_j(d)$ are available.

Lemma 2.3.1 *Dynamic programming for (KP) can be performed in $O(nc)$ time.*

Taking a second look at DP-1 it can be noted that in an iteration of the for–loop with index j only the values $z_{j-1}(d)$ from the previous iteration need to be known to compute $z_j(d)$. All entries $z_k(d)$ with $k < j-1$ from previous iterations are never used again. Hence, it is sufficient to keep two arrays of length c, one containing the values of the recent iteration $j-1$ and the other for the updated values of the current iteration j. In fact, only one array is necessary if the order of the computation is reversed, i.e. if the for–loop is performed for $d := c$ down to $d := w_j$. This yields an improved algorithm denoted by DP-2 and depicted in Figure 2.4. Note that no update is necessary if the currently considered item j is not packed. The space requirement of DP-2 can be bounded by $O(n+c)$ while the running time remains $O(nc)$.

Algorithm DP-2:

```
for d := 0 to c do
    z(d) := 0        initialization
for j := 1 to n do
    for d := c down to w_j do        item j may be packed
        if z(d − w_j) + p_j > z(d) then
            z(d) := z(d − w_j) + p_j
z* := z(c)
```

Fig. 2.4. Basic dynamic programming DP-2 on a single array.

Concerning the computation of the set X^* in DP-2 we can proceed either again in the trivial way of storing c sets $X(d)$ of length n and thus increasing running time and space by a factor of n. Or we carry over the idea of storing pointers $A_j(d)$ as used above. However, to finally reconstruct X^* all the nc entries of this pointer table must be stored which eliminates the improvement for DP-2 and gives again a total time and space complexity of $O(nc)$.

The high memory requirements are frequently cited as a main drawback of dynamic programming. There is an easy way to improve upon the $O(nc)$ space bound for computing both z^* and X^* but at the cost of an increased running time. Reporting the optimal solution set $X^* = X(c)$ at the end without storing the subsets $X(d)$ in every iteration can be done by recording only the most recently added item $r(d)$ for every entry $z(d)$. After the execution of the complete dynamic programming scheme we have at hand item $r(c)$ which is the last item added to the optimal solution. This item clearly belongs to the optimal solution set and the whole dynamic programming procedure (following DP-2) is executed again with the capacity reduced by the weight of item $r(c)$. Instead of going through all items we have to deal only with items whose index number is smaller than $r(c)$ since none of the items with an index larger than $r(c)$ was included in the optimal solution.

The resulting algorithm DP-3 is presented in detail in Figure 2.5. Its space requirement is only $O(n+c)$ while the running time is now $O(n^2c)$ (each of the at most n iterations requires $O(nc)$ time). Note that the commands in the repeat–loop are almost identical to DP-2.

Algorithm DP-3:

$X^* := \emptyset$ *optimal solution set*
$\bar{c} := c$, $\bar{n} := n$
repeat *in every iteration problem $KP_{\bar{n}}(\bar{c})$ is solved by* DP-2
  for $d := 0$ to $\bar{c}$ do
    $z(d) := 0$, $r(d) := 0$ *initialization*
  for $j := 1$ to $\bar{n}$ do
    for $d := \bar{c}$ down to w_j do *item j may be packed*
      if $z(d-w_j) + p_j > z(d)$ then
        $z(d) := z(d-w_j) + p_j$
        $r(d) := j$
  $r := r(\bar{c})$ *new item of the optimal solution set*
  $X^* := X^* \cup \{r\}$
  $\bar{n} := r-1$, $\bar{c} := \bar{c} - w_r$
until $\bar{c} = 0$
$z^* := z(c)$ *computed in the first iteration*

Fig. 2.5. Algorithm DP-3 iteratively executing DP-2 to find the optimal solution set without additional storage.

Fortunately, this deadlock between increasing space (in DP-1 and DP-2) or time (in DP-3) in order to compute X^* can be broken by the application of a general technique described in Section 3.3 (see Corollary 3.3.2). Based on DP-2 the resulting algorithm finds z^* and X^* in $O(nc)$ time while keeping the space requirement of $O(n+c)$.

Reversing the roles of profits and weights we can also perform *dynamic programming by profits*. The main idea of this version is to reach every possible total profit value with a subset of items of minimal total weight. Clearly, the highest total profit value, which can be reached by a subset of weight not greater than the capacity c, will be an optimal solution. This concept is called *dynamic programming by reaching*. It will be used in Section 2.6 to construct a special type of approximation algorithm for the knapsack problem. Since this version of dynamic programming should be easy to understand after the detailed treatment of the previous algorithms in this section, we will only state the necessary definitions and formulas of this approach and give the algorithmic presentation but refrain from a detailed elaboration to avoid repetitions.

Let $y_j(q)$ denote the minimal weight of a subset of items from $\{1,\dots,j\}$ with total profit equal to q. If no such subset exists we will set $y_j(q) := c+1$. To bound the length of every array y_j we have to compute an upper bound U on the optimal solution value. An obvious possibility would be to use the upper bound $U_{\mathrm{LP}} = \lfloor z^{LP} \rfloor$ from the solution of the LP-relaxation (LKP) in Section 2.2 and set $U := U_{\mathrm{LP}}$. Recall from Lemma 2.2.4 that U_{LP} is at most twice as large as the optimal solution value z^*. Initializing $y_0(0) := 0$ and $y_0(q) := c+1$ for $q = 1,\dots,U$, all other values can be computed for $j = 1,\dots,n$ and $q = 0,\dots,U$ by the use of the recursion

$$y_j(q) := \begin{cases} y_{j-1}(q) & \text{if } q < p_j, \\ \min\{y_{j-1}(q),\, y_{j-1}(q-p_j)+w_j\} & \text{if } q \geq p_j. \end{cases} \tag{2.10}$$

The optimal solution value is given by $\max\{q \mid y_n(q) \leq c\}$. A formal description of the resulting algorithm DP-Profits, which is an analogon of DP-2, is given in Figure 2.6.

By analogous arguments as for dynamic programming by weights we get that the running time of DP-Profits is bounded by $O(nU)$. For the space requirements the same discussion applies as before. It can be summarized in the statement that $O(n+U)$ is sufficient to compute z^* but $O(nU)$ is required to find also the corresponding set X^*. Again, the storage reduction scheme which will be presented in Section 3.3 can be easily adapted according to Lemma 2.3.1 thus improving the performance of dynamic programming by profits and yielding a running time of $O(nU)$ and space requirements of $O(n+U)$, where $U \leq 2z^*$.

Lemma 2.3.2 *Let U be an upper bound on the optimal solution value. Then dynamic programming for (KP) can be performed in $O(nU)$ time.*

Algorithm DP-Profits:

compute an upper bound $U \geq z^*$
$y(0) := 0$
for $q := 1$ **to** U **do**
  $y(q) := c+1$ *initialization*
for $j := 1$ **to** n **do**
  for $q := U$ **down to** p_j **do** *item j may be packed*
    if $y(q-p_j)+w_j < y(q)$ **then**
      $y(q) := y(q-p_j)+w_j$
$z^* := \max\{q \mid y(q) \leq c\}$

Fig. 2.6. A version of DP-2 performing dynamic programming by profits.

Example: (cont.) If we run dynamic programming by profits for the items of the example of Section 2.1, we have $c+1 = 10$ and $U = U_{\mathrm{LP}} = 16$. The values of $y_j(q)$ are depicted in Figure 2.7. The index of the largest row with an entry smaller than 10 denotes the optimal solution value z^*. The values in boxes correspond to the weights of the optimal solution set. □

$q \backslash j$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	10	10	10	10	10	10	10	10
2	10	10	10	10	10	10	10	10
3	10	10	10	10	10	10	10	4
4	10	10	10	10	10	10	10	10
5	10	10	3	3	3	3	3	3
6	10	**2**	**2**	**2**	2	2	2	2
7	10	10	10	10	10	10	9	9
8	10	10	10	6	6	6	6	6
9	10	10	10	10	7	7	7	6
10	10	10	10	10	10	10	10	10
11	10	10	5	5	5	5	5	5
12	10	10	10	10	10	7	7	7
13	10	10	10	9	9	9	9	9
14	10	10	10	8	8	8	8	8
15	10	10	10	10	**9**	**9**	**9**	**9**
16	10	10	10	10	10	10	10	10

Fig. 2.7. The values of $y_j(q)$ of the example, denoting the minimal weights of a subset of items of $\{1,\ldots,j\}$ with total profit q.

2.4 Branch-and-Bound

A completely different method than dynamic programming to compute an optimal solution for integer programming problems is the *branch-and-bound* approach. Instead of extending a partial solution iteratively until finally an overall optimal solution is found as in dynamic programming, a brute force algorithm might enumerate all feasible solutions and select the one with the highest objective function value. However, the number of feasible solutions may become extremely large since 2^n different solutions can be generated from n binary variables.

The general algorithmic concept of branch-and-bound is based on an *intelligent* complete enumeration of the solution space since in many cases only a small subset of the feasible solutions are enumerated explicitly. It is however guaranteed that the parts of the solution space which were not considered explicitly cannot contain the optimal solution. Hence, we say that these feasible solutions were enumerated *implicitly*.

As the name indicates, a branch-and-bound algorithm is based on two fundamental principles: *branching* and *bounding*. Assume that we wish to solve the following maximization problem

$$\max_{x \in X} f(x) \tag{2.11}$$

where X is a finite solution space. In the *branching* part, a given subset of the solution space $X' \subseteq X$ is divided into a number of smaller subsets $X_1, \ldots, X_m$. These subsets may in principle be overlapping but their union must span the whole solution space X, thus $X_1 \cup X_2 \cup \ldots \cup X_m = X'$. The process is in principle repeated until each subset contains only a single feasible solution. By choosing the best of all considered solutions according to the present objective value, one is guaranteed to find a global optimum for the stated problem.

The *bounding* part of a branch-and-bound algorithm derives upper and lower bounds for a given subset X' of the solution space. A lower bound $z^\ell \leq z^*$ may be chosen as the best solution encountered so far in the search. If no solutions have been considered yet, one may choose z^ℓ as the objective value of a heuristic solution. An *upper bound* $U_{X'}$ for a given solution space $X' \subseteq X$ is a real number satisfying

$$U_{X'} \geq f(x), \quad \text{for all } x \in X'. \tag{2.12}$$

The upper bound is used to prune parts of the search space as follows. Assume that $U_{X'} \leq z^\ell$ for a given subset X'. Then (2.12) immediately gives that

$$f(x) \leq U_{X'} \leq z^\ell, \quad \text{for all } x \in X', \tag{2.13}$$

meaning that a better solution than z^ℓ cannot be found in X' and thus we need not investigate X' further.

Although an upper bound can be derived as $U_{X'} = \max_{x \in X'} f(x)$, this approach is computationally too expensive to be applied. Instead one may extend the solution space, or modify the objective function such that the hereby obtained problem

can be solved efficiently. An extended solution space $Y \supseteq X'$ is obtained by *relaxing* some of the constraints to the considered problem. If Y is chosen properly, $U_{X'} = \max_{x \in Y} f(x)$ may be derived efficiently, and it is easily seen that the derived value is indeed an upper bound. The other approach is to consider a different objective function $g(x)$ which satisfies that $g(x) \geq f(x)$ for all $x \in X'$. As before it is easily verified that $U_{X'} = \max_{x \in X'} g(x)$ gives a valid upper bound. The two approaches, *relaxation* and *modification of the objective value*, are often combined when deriving upper bounds.

We will apply branch-and-bound to (KP) in the following, having

$$f(x) = \sum_{j=1}^{n} p_j x_j, \tag{2.14}$$

$$X = \left\{ x_j \in \{0,1\} \,\middle|\, \sum_{j=1}^{n} w_j x_j \leq c \quad j = 1,\ldots,n \right\}. \tag{2.15}$$

An upper bound is derived by dropping the integrality constraint in (2.15) such that we get

$$Y = \left\{ 0 \leq x_j \leq 1 \,\middle|\, \sum_{j=1}^{n} w_j x_j \leq c \quad j = 1,\ldots,n \right\}. \tag{2.16}$$

The latter problem is recognized as the linear programming relaxation (LKP) of (KP) considered in Section 2.2 which can be derived in $O(n)$ time if we initially sort the items according to decreasing efficiencies (2.2) or even without sorting if we apply the median algorithm (see Lemma 3.1.1 of Section 3.1).

The branching operation may easily be implemented by splitting the solution space X' in two parts X'_1 and X'_2, where $x_j = 1$ in X_1 and $x_j = 0$ in X_2 for a given variable x_j.

The resulting procedure Branch-and-Bound first initializes z e.g. to the greedy solution and x' to the zero vector.

To make things simple, a presented branch-and-bound algorithm can be written as a recursive procedure such that we make use of the tree-search facilities supported by the programming language. In Figure 2.8 we have depicted subprocedure Branch-and-Bound(ℓ) in which branching is performed for variable x_ℓ and Branch-and-Bound($\ell + 1$) is called. Procedure Branch-and-Bound(1) corresponds to Branch-and-Bound after initialization. Those variables for which branching is not performed yet, are often called *free variables*. The process of branch-and-bound is illustrated in a rooted tree which we call the *branch-and-bound tree*. The nodes of the branch-and-bound tree correspond to subsets X' of the solution space and the edges leaving a node correspond to the divisions of X' into smaller subsets.

Algorithm Branch-and-Bound(ℓ):

ℓ is the index of the next variable to branch at
the current solution is $x_j = x'_j$ for $j = 1, \ldots, \ell - 1$
the current solution subspace is
$X' = \{x_j \in \{0,1\} \mid \sum_{j=1}^{\ell-1} w_j x'_j + \sum_{j=\ell}^{n} w_j x_j \leq c\}$

if $\sum_{j=1}^{\ell-1} w_j x'_j > c$ then return *X' is empty*
if $\sum_{j=1}^{\ell-1} p_j x'_j > z$ then
$z := \sum_{j=1}^{\ell-1} x'_j$, $x^* := x'$ *improved solution z*
if ($\ell > n$) then return *X' contains only the optimal solution*
derive $U_{X'} = \max_{x \in Y} f(x)$
where $Y = \{0 \leq x_j \leq 1 \mid \sum_{j=1}^{\ell-1} w_j x'_j + \sum_{j=\ell}^{n} w_j x_j \leq c\}$
if $u_{X'} > z$ then
$x'_\ell := 1$, Branch-and-Bound($\ell + 1$)
$x'_\ell := 0$, Branch-and-Bound($\ell + 1$)

Fig. 2.8. Branch-and-Bound(ℓ) branching for variable x_ℓ and calling Branch-and-Bound($\ell + 1$).

The recursive structure of Branch-and-Bound will result in a so-called *depth-first search*. A different search strategy could be *best-first search* where we always investigate the subproblem with largest upper bound first, hoping that this will lead more quickly to an improved lower bound. Best first search however demands some explicit data structure to store the subproblems considered, and the space consumption may become exponentially large.

Example: (cont.) The application of Branch-and-Bound to our example of Section 2.1 is illustrated in the branch-and-bound tree of Figure 2.9. Lower bounds are marked by underlined numbers and upper bounds by overlined numbers, respectively. The lower bounds are found by algorithm Greedy. The upper bounds are the values of U_{LP}, i.e. they are calculated by rounding down the solution of the linear programming relaxation (LKP). Of course, both bounds are recalculated depending on the variables which have been fixed already. We omitted branching when the corresponding item could not be packed into the knapsack since the capacity restriction was violated and proceed with the next node for which a packing of an item is possible. The bounds for the optimal solution are written in bold. □

2.5 Approximation Algorithms

In the previous sections two different approaches to compute optimal solutions of the knapsack problem were presented. Although only the most basic versions of the

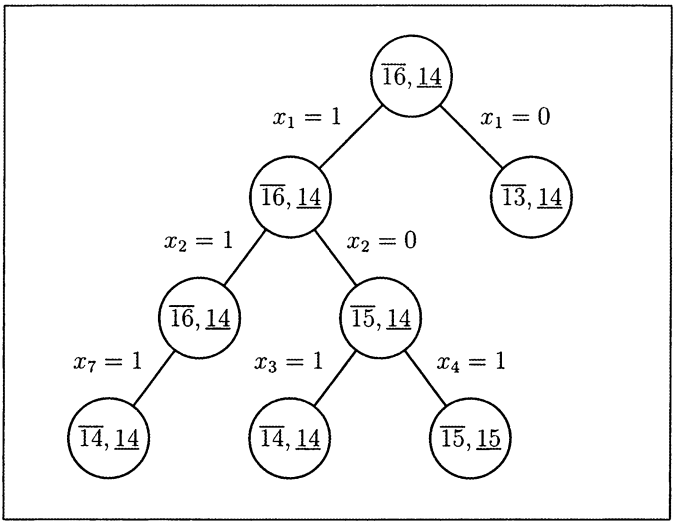


Fig. 2.9. A tree illustrating Branch-and-Bound applied to our example.

two methods were introduced and many improvements with considerable practical impact are known, there remain serious drawbacks for both of them with respect to their practical performance.

All versions of dynamic programming for the knapsack problem are pseudopolynomial algorithms, i.e. both the running time and the memory requirements are dependent on the size of the capacity c or any profit bound U which may be unpleasantly large even for "easy" problems with only a small number of items depending on the field of application.

The running time complexity of pure branch-and-bound algorithms usually cannot be described explicitly since the actual number of nodes in the branch-and-bound tree cannot be bounded a priori. Indeed, there are particular instances of (KP) which are extremely time consuming to solve for any branch-and-bound algorithm (see Section 5.5.1). Combining branch-and-bound with dynamic programming ideas leads again to algorithms with pseudopolynomial running time (see Section 3.5).

These attempts to compute optimal solutions were not completely satisfying, especially if the available resources of time and space are very limited. As mentioned in Section 1.5 this negative aspect is not caused by missing a more clever approach to the problem but is indeed a general property of the knapsack problem. Consequently, we have to look for other ways to deal with the situation. In particular, in many applications of the knapsack problem it is not really necessary to find an optimal solution of (KP) at all costs but also a "good" solution would be fine enough, especially if it can be computed in reasonable time. In some sense, we have to pay

for the reduced running time by getting only a suboptimal solution. This point of view is the basis of *approximation algorithms*.

In principle, any algorithm which computes a feasible solution of (KP) is an approximation algorithm generating approximate solutions. Of course, we would like to attain approximate solutions which are not too far away from the optimal solution. Moreover, we would be interested to get some information about the size of this deviation from optimality. One possible measure for this is the investigation of the *expected performance* of an algorithm under a probabilistic model. In particular the expected difference of the approximate from the optimal solution would be worth to be determined. Unfortunately, only very simple algorithms can be tackled successfully by this approach. Naturally, results in this direction apply only under special stochastic conditions. Moreover, these results are valid only if the number of items is extremely large or approaches infinity and thus yield limited insight for practical problems. The topic will be further elaborated on in Chapter 14.

Simulation studies, where an approximation algorithm is executed many times with different sets of input data generated under a stochastic model, can give a reasonable picture of the practical behaviour of an algorithm but of course only for instances following the assumptions of the data model. Therefore, simulation cannot provide a general certificate about the quality of an approximation method but only a reasonable guess for the behaviour of the algorithm for a particular class of problem instances.

In general it may happen that an approximation algorithm sometimes generates solutions with an almost optimal value but in other cases produces inferior solutions. Therefore, we will investigate the *worst-case behaviour* of an approximation algorithm, namely the largest possible distance over all problem instances between the approximate solution and an optimal solution. There are two reasonable ways to measure this distance.

The most natural idea to measure such a distance is the *absolute performance guarantee* which is determined by the maximum difference between the optimal solution value and the approximate solution value over all problem instances. Let us define the optimal solution value of a problem instance I as $z^*(I)$ and the solution value computed by an approximation algorithm A as $z^A(I)$. The reference to the instance I will later be frequently omitted and we will write briefly z^* respectively z^A if there is no ambiguity. Note that we will only deal with the approximation of problems where the objective is to *maximize* and hence $z^A(I) \leq z^*(I)$.

Definition 2.5.1 *An algorithm A is an approximation algorithm with* absolute performance guarantee k, $k > 0$, *if*

$$z^*(I) - z^A(I) \leq k$$

holds for all problem instances I.

Unfortunately, there is only a negative result concerning absolute performance guarantees for the knapsack problem. Indeed, the following theorem says that if an approximation algorithm would have an absolute performance guarantee k for some fixed k, then this algorithm could be used to solve any instance of a knapsack problem to optimality.

Let $f(n,c)$ be a function describing the running time of an algorithm for a knapsack problem with n items and capacity c. Then the following holds:

Theorem 2.5.2 *Let A be an approximation algorithm for (KP) with absolute performance guarantee k and running time in $O(f(n,c))$ for any instance with n items and capacity c. Then A also computes the optimal solution of any such instance in $O(f(n,c))$ time.*

Proof. The statement will be shown by a *scaling procedure* which transfers any given instance I of a knapsack problem into a new instance I' such that any feasible solution of I', which differs not too much from its optimal solution, is in fact an optimal solution for I.

Consider an approximation algorithm A for (KP) with absolute performance guarantee k and let I be given by $p_1,\ldots,p_n,w_1,\ldots,w_n,c$. We construct the new instance I' with $p'_1,\ldots,p'_n,w'_1,\ldots,w'_n,c'$ such that $p'_j=(k+1)p_j$, $w'_j=w_j$ for $j=1,\ldots,n$ and $c'=c$. Clearly, both instances I and I' have exactly the same set of feasible solutions. Since the objective function value of a solution of I' is exactly $k+1$ times as large as the objective function value of the same solution for I, the two instances also have the same optimal solutions. Applying algorithm A to the instance I' yields a solution $z^A(I')$ with

$$z^*(I')-z^A(I')\leq k$$

using the absolute performance guarantee of A. Interpreting the output of A as a solution for I yields a feasible solution with value $z^A(I)$ where $z^A(I')=(k+1)z^A(I)$. Moreover, we also have for the optimal solution value $z^*(I')=(k+1)z^*(I)$. Inserting in the above inequality yields

$$(k+1)z^*(I)-(k+1)z^A(I)\leq k \Longleftrightarrow z^*(I)-z^A(I)\leq \frac{k}{k+1}.$$

But since all profit values are integer, also $z^*(I)-z^A(I)$ must be integer and hence this inequality implies

$$z^*(I)-z^A(I)\leq \left\lfloor \frac{k}{k+1}\right\rfloor \Longrightarrow z^*(I)-z^A(I)\leq 0 \Longrightarrow z^*(I)=z^A(I).$$

Since the approximate solution of I' was computed in $O(f(n,c))$ running time an optimal solution of I could be constructed within the same time. □

Theorem 2.5.2 indicates that an algorithm with fixed absolute performance guarantee would have the same unpleasant running time as an optimal algorithm. As indicated in Section 1.5 and shown formally in Appendix A, it is very unlikely that an

optimal algorithm for (KP) with a running time polynomial in n exists. Hence, also approximation algorithms with fixed absolute performance ratio are bound to have a pseudopolynomial running time which is unpleasantly dependent on the magnitude of the input values.

The second reasonable way to measure the distance between an approximate and an optimal solution is the *relative performance guarantee* which basically bounds the maximum ratio between the approximate and an optimal solution. This expression of the distance as percentage of the optimal solution value seems to be more plausible than the absolute performance guarantee since it is not dependent on the order of magnitude of the objective function value.

Definition 2.5.3 *An algorithm A is an approximation algorithm with* relative performance guarantee k *if*

$$\frac{z^A(I)}{z^*(I)} \geq k$$

holds for all problem instances I.

Clearly, this definition of an approximation algorithm only makes sense for $0 < k < 1$. Recall that we only deal with maximization problems. Frequently, an algorithm with relative performance guarantee k will be called a *k–approximation algorithm* and k is called the *approximation ratio*. Putting emphasis on the relative difference between approximate and optimal solution value we can define $\varepsilon := 1 - k$ and thus refer to a *$(1-\varepsilon)$–approximation algorithm* where

$$\frac{z^A(I)}{z^*(I)} \geq 1 - \varepsilon \quad \Longleftrightarrow \quad \frac{z^*(I) - z^A(I)}{z^*(I)} \leq \varepsilon$$

holds for all problem instances I.

To indicate whether the analysis of the approximation ratio of an algorithm can be improved or not we say that an approximation ratio k is *tight* if no larger value $k' > k$ exists such that k' is also a relative performance guarantee for the considered algorithm. This tightness is usually proved by an example, i.e. a problem instance I where the bound is achieved or by a sequence of instances, where the ratio $\frac{z^A(I)}{z^*(I)}$ gets arbitrarily close to k.

In contrast to the absolute approximation measure the results for relative approximation algorithms are plentiful. Going back to algorithm Greedy from Section 2.1 we might try to establish a reasonable approximation ratio for this simple approach. However, there is no such luck. Consider the following instance of (KP):

Let $n = 2$ and $c = M$. Item 1 is given by $w_1 = 1$ and $p_1 = 2$ whereas item 2 is defined by $w_2 = p_2 = M$. Comparing the efficiencies of the items we get $e_1 = 2$ and $e_2 = 1$. Hence, Greedy starts by packing item 1 and is finished with an approximate

solution value of 2 whereas the optimal solution would pack item 2 thus reaching an optimal solution value of M. For a suitably large selection of M any relative performance guarantee for Greedy can be forced to be arbitrarily close to 0.

However, a small modification of Greedy can avoid this pathological special case. Therefore we define the extended greedy algorithm Ext-Greedy which consists of running Greedy first and then comparing the solution with the highest profit value of any single item. The larger of the two is finally taken as the solution produced by Ext-Greedy. Denoting as before the solution value of Greedy respectively Ext-Greedy by z^G respectively z^{eG}, we simply add the following command at the end of algorithm Greedy:

$$z^{eG} := \max\{z^G, \max\{p_j \mid j = 1, \ldots, n\}\} \tag{2.17}$$

The running time of the extended algorithm is still $O(n)$ if the items are already sorted according to their efficiency. It can be shown quite easily that this new algorithm Ext-Greedy is a $\frac{1}{2}$–approximation algorithm.

Theorem 2.5.4 *Algorithm* Ext-Greedy *has a relative performance guarantee of* $\frac{1}{2}$ *and this bound is tight.*

Proof. From Corollary 2.2.3 we know

$$z^* \leq z^G + p_{\max} \leq z^{eG} + z^{eG} = 2z^{eG}$$

which proves the main part of Theorem 2.5.4.

It can be shown that $\frac{1}{2}$ is also a tight performance ratio for Ext-Greedy by considering the following example of a knapsack problem which is a slight modification of the instance used for Greedy.

Let $n = 3$ and $c = 2M$. Again item 1 is given by $w_1 = 1$ and $p_1 = 2$ and items 2 and 3 are identical with $w_2 = p_2 = w_3 = p_3 = M$. The efficiencies of the items are given by $e_1 = 2$ and $e_2 = e_3 = 1$. Algorithm Ext-Greedy packs items 1 and 2 reaching an approximate solution value of $2 + M$ whereas the optimal solution would pack items 2 and 3 reaching an optimal solution value of $2M$. Choosing M suitably large the ratio between approximate and optimal solution value can be arbitrarily close to $\frac{1}{2}$. □

It should be noted that the same relative performance guarantee of $\frac{1}{2}$ of Ext-Greedy still holds even if the greedy part is replaced by algorithm Greedy-Split introduced in Section 2.1. This solution can be computed in $O(n)$ time even without sorting the items by their efficiency because the split item can be found in $O(n)$ time by the method described in Section 3.1.

Getting a solution which is in the worst-case only half as good as the optimal solution does not sound very impressive. Hence, it is worthwhile trying to find a better approximation algorithm, possibly at the cost of an increased running time.

In the previous worst-case example Ext-Greedy made the "mistake" of excluding item 3 which had a very high profit value. To avoid such a mistake the following algorithm $G^{\frac{3}{4}}$ (the terminology follows from the subsequent result that this algorithm has approximation ratio $\frac{3}{4}$) considers not only every single item as a possible solution (naturally taking the one with the highest profit) but also goes through all feasible *pairs of items*. This can be seen as "guessing" the two items with largest profit in the optimal solution. In fact by going through all pairs of items also this particular pair must be considered at some point of the execution. For each generated pair the algorithms fills up the remaining capacity of the knapsack by Ext-Greedy using only items with profit smaller than or equal to the profits of the two selected items.

A detailed formulation is given in Figure 2.10. We do not include the computation of the item set corresponding to every considered feasible solution since this point should follow trivially from the construction.

Algorithm $G^{\frac{3}{4}}$:
  $z^A := \max\{p_j \mid j = 1,\ldots,n\}$
  z^A *is the value of the currently best solution*
  for all pairs of items $(i,k) \in N \times N$ do
    if $w_i + w_k \leq c$ then
      apply algorithm Ext-Greedy to the knapsack instance with
      item set $\{j \mid p_j \leq \min\{p_i, p_k\}\} \setminus \{i,k\}$ and capacity $c - w_i - w_k$ returning solution value $\bar{z}$.
      if $p_i + p_k + \bar{z} > z^A$ then $z^A := p_i + p_k + \bar{z}$
      a new currently best solution is found

Fig. 2.10. Algorithm $G^{\frac{3}{4}}$ guessing the two items with largest profit.

Theorem 2.5.5 *Algorithm* $G^{\frac{3}{4}}$ *has a tight relative performance guarantee of* $\frac{3}{4}$.

Proof. As will be frequently the case for approximation algorithms the proof of the theorem is performed by comparing the solution z^A computed by $G^{\frac{3}{4}}$ with an optimal solution z^*. If z^* is generated by a single item, there will clearly be $z^A = z^*$ after the first line of $G^{\frac{3}{4}}$. Otherwise consider the two items i^*, k^* which contribute the highest profit in the optimal solution. Clearly, $G^{\frac{3}{4}}$ generates this pair at some point of its execution and applies Ext-Greedy to the knapsack problem remaining after packing i^* and k^*. Denote by z_S^* the optimal solution of the corresponding subproblem S with item set $\{j \mid p_j \leq \min\{p_{i^*}, p_{k^*}\}\} \setminus \{i^*, k^*\}$ and capacity $c - w_{i^*} - w_{k^*}$ such that $z^* = z_S^* + p_{i^*} + p_{k^*}$. Also denote the approximate solution computed by Ext-Greedy by z_S^{eG}. It follows from the construction of $G^{\frac{3}{4}}$ that $z^A \geq p_{i^*} + p_{k^*} + z_S^{eG}$ and from Theorem 2.5.4 that $z_S^{eG} \geq \frac{1}{2} z_S^*$. Now there are two cases to be considered.

Case I: $p_{i^*} + p_{k^*} \geq \frac{1}{2}z^*$
From above we immediately get

$$z^A \geq p_{i^*} + p_{k^*} + z_S^{eG} \geq p_{i^*} + p_{k^*} + \frac{1}{2}z_S^*.$$

With the condition of Case I this can be tranformed to

$$z^A \geq p_{i^*} + p_{k^*} + \frac{1}{2}(z^* - p_{i^*} - p_{k^*}) = \frac{1}{2}(z^* + p_{i^*} + p_{k^*}) \geq \frac{1}{2}(z^* + \frac{1}{2}z^*) = \frac{3}{4}z^*.$$

Case II: $p_{i^*} + p_{k^*} < \frac{1}{2}z^*$
Clearly at least one of the two items i^*, k^* must have a profit smaller than $\frac{1}{4}z^*$. Hence, by definition z_S^* consists only of items with profit smaller than $\frac{1}{4}z^*$. But then we get for the LP-relaxation of subproblem S with value z_S^{LP} from Corollary 2.2.2 by adding the complete split item s'

$$z_S^* \leq z_S^{LP} \leq z_S^{eG} + p_{s'} \leq z_S^{eG} + \frac{1}{4}z^*.$$

This can be put together resulting in

$$z^* = p_{i^*} + p_{k^*} + z_S^* \leq p_{i^*} + p_{k^*} + z_S^{eG} + \frac{1}{4}z^* \leq z^A + \frac{1}{4}z^*.$$

The tightness of the performance bound of $\frac{3}{4}$ for $G^{\frac{3}{4}}$ is shown by the following example.

Example: Let $n = 5$ and $c = 4M$. Item 1 is given by $w_1 = 1$ and $p_1 = 2$ and items 2 to 5 are all identical with $w_2 = \ldots = w_5 = M$ and $p_2 = \ldots = p_5 = M$. These items are called "big items". A pair of items consists either of two big items, which means that Ext-Greedy returns item 1 and another big item as solution of the subproblem resulting in a total solution value of $3M + 2$, or a pair consists of item 1 and one big item. In the latter case no items remain for Ext-Greedy to pack. Clearly, the optimal solution consists of four big items with total profit $4M$ and the ratio between approximate and optimal solution value can be arbitrarily close to $\frac{3}{4}$ for suitably large M. □

The running time of $G^{\frac{3}{4}}$ is basically determined by executing Ext-Greedy, which requires $O(n)$ time, for every pair of items. This number of executing Ext-Greedy is $\binom{n}{2}$ which is in $O(n^2)$ and hence the total running time is $O(n^3)$. It will be shown in Section 6.1 that this running time can be reduced to $O(n^2)$ in a more clever implementation of $G^{\frac{3}{4}}$. The total space requirements are only $O(n)$.

2.6 Approximation Schemes

The idea in algorithm $G^{\frac{3}{4}}$ from the previous section of going through all feasible pairs of items can be extended in a natural way to all triples, quadruples or k-tuples of items. In this way the relative performance ratio is further improved but also the running time increases considerable. This relation which assigns for every k a relative performance ratio to an approximation algorithm testing all k-tuples of items can also be seen in an inverse way. We are given a relativ performance ratio which should be attained and are asked to select an appropriate k for running the algorithm with all k-tuples of items. This task of reaching a given relative performance ratio is very natural since in practice one is interested in the quality of the approximate solution and not in algorithmic parameters.

This motivates to introduce the concept of an ε–*approximation scheme*. As before, we concentrate on the knapsack family and hence deal with maximization problems only.

Definition 2.6.1 *An algorithm A is an* ε–approximation scheme *if for every input* $\varepsilon \in]0,1[$

$$z^A(I) \geq (1-\varepsilon)z^*(I)$$

holds for all problem instances I.

This is equivalent to the statement that A is a $(1-\varepsilon)$–approximation algorithm for every input value ε. It is quite natural that usually the running time of an ε–approximation scheme will increase with decreasing ε. The "better" the solution value, the higher the necessary running time. Later in this section we will introduce classes of approximation schemes where this increase of the running time is restricted in some way.

Generalizing the above algorithm $G^{\frac{3}{4}}$ we derive the following ε–approximation scheme H^{ε} for the knapsack problem. Instead of taking the single item with highest profit as a first solution we try all sets of items which have a cardinality smaller than some fixed parameter ℓ depending on ε, of course. Then we try to "guess" the ℓ items in the optimal solution with the highest profits. This is done by going through all sets of exactly ℓ items and trying to pack them. If this is possible we fill the remaining knapsack by Ext-Greedy using only items with smaller profit.

To avoid trivial cases we will assume that $\varepsilon \leq \frac{1}{2}$. Note that for $\varepsilon = \frac{1}{2}$ the algorithm is equivalent to Ext-Greedy. Moreover it is easy to check that for the case of $\varepsilon = \frac{1}{4}$ algorithm H^{ε} is equivalent to $G^{\frac{3}{4}}$ with $\ell = 2$.

Note that the separation of the cases $|L| < \ell$ and $|L| = \ell$ is not really necessary for the correctness of H^{ε} but will be convenient in the more complicated version described in Section 6.1.

Algorithm H^ε :

$\ell := \min\{\lceil \frac{1}{\varepsilon} \rceil - 2, n\}$
$z^A := 0$ $\quad z^A$ *is the value of the currently best solution*
for all subsets $L \subset N$ with $|L| \leq \ell - 1$ do
  if $\sum_{j \in L} w_j \leq c$ then
    if $\sum_{j \in L} p_j > z^A$ then
      $z^A := \sum_{j \in L} p_j$ $\quad$ *a new currently best solution*
for all subsets $L \subseteq N$ with $|L| = \ell$ do
  if $\sum_{j \in L} w_j \leq c$ then
    apply algorithm Ext-Greedy to the knapsack instance with item set $N := \{j \mid p_j \leq \min\{p_i \mid i \in L\}\} \setminus L$ and capacity $c - \sum_{j \in L} w_j$ returning solution value $\bar{z}$.
    if $\sum_{j \in L} p_j + \bar{z} > z^A$ then
      $z^A := \sum_{j \in L} p_j + \bar{z}$ *a new currently best solution*

Fig. 2.11. The ε–approximation scheme H^ε for the knapsack problem.

Theorem 2.6.2 *Algorithm* H^ε *is an* ε*–approximation scheme.*

Proof. We will proceed in a completely analogous way to the proof of Theorem 2.5.5. If the optimal solution z^* consists of less than ℓ items we will enumerate this solution at some point during the execution of the first for–loop. Otherwise let us consider the set L^* containing the ℓ items which contribute the highest profit in the optimal solution. Algorithm H^ε generates this set during the execution of the second for–loop and applies Ext-Greedy to the knapsack problem remaining after packing all items in L^*. Denote again by z_S^* the optimal solution of the corresponding subproblem S with item set $N := \{j \mid p_j \leq \min\{p_i \mid i \in L^*\}\} \setminus L^*$ and capacity $c - \sum_{j \in L^*} w_j$ and the approximate solution computed by Ext-Greedy by z_S^{eG}, respectively. As before we have for H^ε that $z^A \geq \sum_{j \in L^*} p_j + z_S^{eG}$ and from Theorem 2.5.4 that $z_S^{eG} \geq \frac{1}{2} z_S^*$. Again there are two cases to be considered.

Case I: $\sum_{j \in L^*} p_j \geq \frac{\ell}{\ell+2} z^*$

From above we immediately get

$$z^A \geq \sum_{j \in L^*} p_j + z_S^{eG} \geq \sum_{j \in L^*} p_j + \frac{1}{2} z_S^*.$$

The condition of Case I yields

$$z^A \geq \sum_{j \in L^*} p_j + \frac{1}{2}\left(z^* - \sum_{j \in L^*} p_j\right) = \frac{1}{2}\left(z^* + \sum_{j \in L^*} p_j\right)$$
$$\geq \frac{1}{2}\left(z^* + \frac{\ell}{\ell+2} z^*\right) = \frac{\ell+1}{\ell+2} z^*.$$

Case II: $\sum_{j \in L^*} p_j < \frac{\ell}{\ell+2} z^*$

At least one of the ℓ items in L^* must have a profit smaller than $\frac{1}{\ell+2} z^*$. By definition z_S^* consists only of items with profit smaller than $\frac{1}{\ell+2} z^*$. For the LP-relaxation of subproblem S with value z_S^{LP} we get from Theorem 2.2.1 by adding the complete split item s'

$$z_S^* \leq z_S^{LP} \leq z_S^{eG} + p_{s'} \leq z_S^{eG} + \frac{1}{\ell+2} z^*.$$

This can be put together resulting in

$$z^* = \sum_{j \in L^*} p_j + z_S^* \leq \sum_{j \in L^*} p_j + z_S^{eG} + \frac{1}{\ell+2} z^* \leq z^A + \frac{1}{\ell+2} z^*.$$

Since $\frac{\ell+1}{\ell+2}$ is increasing with ℓ we get from the definition of ℓ

$$\frac{\ell+1}{\ell+2} \geq \frac{\frac{1}{\varepsilon} - 1}{\frac{1}{\varepsilon}} = 1 - \varepsilon$$

and thus have shown that in both cases $z^A \geq (1-\varepsilon) z^*$. □

Also the "bad" example from above can be generalized to show that H^ε can reach a relative performance ratio quite near to $1 - \varepsilon$.

Let $n = \lceil \frac{1}{\varepsilon} \rceil + 1$ and $c = \lceil \frac{1}{\varepsilon} \rceil M$. Item 1 is given by $w_1 = 1$ and $p_1 = 2$ and items 2 to n are all identical with $w_2 = \ldots = w_n = M$ and $p_2 = \ldots = p_n = M$. These items are again called "big items" as in the proof of Theorem 2.5.5. Any subset of $\ell = n - 3$ items either contains only big items in which case Ext-Greedy returns item 1 and another big item as solution of the subproblem yielding a total solution value of $(\ell+1)M + 2$, or it contains item 1 and $\ell - 1$ big items. In the latter case Ext-Greedy packs another two big items which leads to the same solution as before. However, the optimal solution consists of all $\ell + 2$ big items with total profit $(\ell + 2)M$. The ratio between approximate and optimal solution value converges to $\frac{\ell+1}{\ell+2}$ for increasing M.

Considering the running time of H^ε there are mainly the $\binom{n}{\ell}$ (trivially bounded by $O(n^\ell)$) executions of Ext-Greedy for all possible subsets of cardinality ℓ to perform. Since each of them requires linear time this yields an $O(n^{\ell+1})$ running time bound. In Theorem 6.1.1 of Section 6.1 this time bound will be reduced to $O(n^\ell)$ by a better implementation of H^ε. For the sake of completeness we also have to mention that Ext-Greedy requires an additional $O(n \log n)$ time for sorting the items (of course only once) if it is not replaced by the simplified version with running time $O(n)$ mentioned in the previous section. Note that beside storing the input data no significant additional memory is used and the total space requirements are thus only $O(n)$.

Algorithm H^ε is a straightforward method and quite easy to implement. However, if the desired accuracy gets higher its running time basically explodes. Consider that even for a moderate relative performance guarantee of 0.9, i.e. $\varepsilon = \frac{1}{10}$, which means that a deviation of 10 percent from the optimal solution value is still acceptable, we have a running time of $O(n^9)$ or still $O(n^8)$ in the improved implementation.

Obviously, it would be desirable to have ε–approximation schemes where the running time increases only moderately *both* with the number of items *and* with the accuracy. This discussion leads to a classification of ε–approximation schemes depending on the influence of ε on their running time. In particular, we will distinguish whether the running time is polynomial in $\frac{1}{\varepsilon}$ or whether ε may appear also in the exponent of the running time bound. More formal definitions in the sense of theoretical computer science can be found in the books by Hochbaum [233] and Ausiello et al. [14].

Definition 2.6.3 *An ε–approximation scheme A is a* polynomial time approximation scheme *(*PTAS*) if its running time is polynomial in n.*

This means that the running time increases only polynomially with the number of items but may "explode", i.e. increase exponentially if the required accuracy gets higher, e.g. as in the previous algorithm H^ε . We immediately get from Theorem 2.6.2 and the above analysis.

Theorem 2.6.4 *Algorithm H^ε is a* PTAS *for the knapsack problem.*

Including the accuracy in the polynomiality of the running time we can define a more advanced class of approximation schemes.

Definition 2.6.5 *An ε–approximation scheme A is a* fully polynomial time approximation scheme *(*FPTAS*) if its running time is polynomial both in n and in $\frac{1}{\varepsilon}$.*

Indeed, there exist also fully polynomial approximation schemes for the knapsack problem. It should be noted that in many of the known *FPTAS*s the improved time complexity is paid for by a considerable increase in space requirement which also depends on n and $\frac{1}{\varepsilon}$. Hence, it was frequently pointed out that the use of an *FPTAS* for the knapsack problem is impractical (see Martello and Toth [335, page 72]). However, the recent *FPTAS* by Kellerer and Pferschy [267, 266], which will be described in Section 6.2, decreases the space complexity to $O(n + 1/\varepsilon^2)$ thus yielding an approximation scheme with higher practical relevance.

In general, the construction of an *FPTAS* requires more technical tools. In this section we will only describe a very simple *FPTAS* to illustrate the basic approach to construct fully polynomial approximation schemes. Later on in Section 6.2 much more complicated schemes with a considerably improved running time and space requirements will be presented.

Our approach will rely on an appropriate *scaling* of the profit values p_j. With these new profits we simply run DP-Profits as introduced at the end of Section 2.3. Scaling means here that the given profit values p_j are replaced by new profits $\tilde{p}_j$ such that $\tilde{p}_j := \lfloor \frac{p_j}{K} \rfloor$ for a constant K which will be chosen later.

Note that scaling the weights instead of the profits would not result in a correct algorithm since rounding down the item weights would generate infeasible solutions where the total weight of the original values is larger than the capacity, whereas rounding up the weights might easily make infeasible all solutions which are near to the optimum. As an example consider the case where all items have weight slightly less than $c/2$ and are rounded up to a value slightly larger than $c/2$.

This scaling can be seen as a partitioning of the profit range into intervals of length K with starting points $0, K, 2K, \ldots$. Naturally, for every profit value p_j there is some integer value $i \geq 0$ such that p_j falls into the interval $[iK, (i+1)K[$. The scaling procedure generates for every p_j the value $\tilde{p}_j$ as the corresponding index i of the lower interval bound iK.

Running dynamic programming by profits yields a solution set $\tilde{X}$ for the scaled items which will usually be different from the original optimal solution set X^*. Evaluating the original profits of item set $\tilde{X}$ yields the approximate solution value z^A. The difference between z^A and the optimal solution value can be bounded as follows.

$$\begin{aligned} z^A &= \sum_{j \in \tilde{X}} p_j \geq \sum_{j \in \tilde{X}} K \left\lfloor \frac{p_j}{K} \right\rfloor \geq \sum_{j \in X^*} K \left\lfloor \frac{p_j}{K} \right\rfloor \geq \sum_{j \in X^*} K \left(\frac{p_j}{K} - 1 \right) = \\ &= \sum_{j \in X^*} (p_j - K) = z^* - |X^*|K. \end{aligned} \tag{2.18}$$

The crucial second inequality is due to the fact that the optimal solution value of the scaled instance will always be at least as large as the scaled profits of the items of the original optimal solution.

To get the desired performance guarantee of $1 - \varepsilon$ there must be

$$\frac{z^* - z^A}{z^*} \leq \frac{|X^*|K}{z^*} \leq \varepsilon.$$

Hence, we have to choose K such that

$$K \leq \frac{\varepsilon z^*}{|X^*|}. \tag{2.19}$$

Since $n \geq |X^*|$ and $p_{\max} \leq z^*$ the right-hand side of (2.19) can be bounded by

$$\frac{\varepsilon z^*}{|X^*|} \geq \frac{\varepsilon z^*}{n} \geq \frac{\varepsilon p_{\max}}{n}.$$

Therefore, choosing $K := \varepsilon \frac{p_{\max}}{n}$ clearly satisfies condition (2.19) and thus guarantees the performance ratio of $1 - \varepsilon$.

Given an upper bound U on the optimal solution value and considering the running time of this algorithm, we have to "transform" the $O(n^2U)$ bound for dynamic programming by profits of the scaled problem instance (see Lemma 2.3.2). The optimal solution value of the scaled problem $\tilde{z}$ is clearly bounded by $\tilde{z} \leq n\tilde{p}_{\max}$. Since $\tilde{p}_{\max} \leq \frac{p_{\max}}{K} = \frac{n}{\varepsilon}$, we trivially get $\tilde{z} \leq \frac{n^2}{\varepsilon}$. This bound on the optimal solution value can be substituted for U in the running time bound and yields an overall running time of $O(n^3 \frac{1}{\varepsilon})$ which is polynomial both in n and in $\frac{1}{\varepsilon}$. The space requirements are given by $O(n^3 \frac{1}{\varepsilon})$. Both of these complexity bounds will be improved considerably in Section 6.2. Summarizing we state the following theorem.

Theorem 2.6.6 *There is an* FPTAS *for the knapsack problem.*

3. Advanced Algorithmic Concepts

There are several more technical algorithmic aspects which are useful tools for many of the algorithms presented throughout this book. Although some of them may be seen only as implementational details, others are nice algorithmic and structural ideas on their own. Both of these categories are frequently crucial to derive the claimed running time bounds. We will try to present them as "black boxes" which means that we state explicitly a certain algorithmic property or a feature to be used by algorithms thereafter. The corresponding detailed description or justification is given for the interested reader but may be omitted without loss of understanding in the remaining chapters.

3.1 Finding the Split Item in Linear Time

Both the linear programming relaxation (LKP) of Section 2.2 and the $\frac{1}{2}$-approximation algorithm Ext-Greedy of Section 2.1 respectively 2.5 are dependent on the split item s. Recall from Section 2.5 that we attain the same $\frac{1}{2}$-approximation ratio if we stop Ext-Greedy after failing to pack an item (i.e. the split item) for the first time, i.e. replacing Greedy by Greedy-Split. Clearly, it is trivial to find s in $O(n)$ time if the items are already sorted in decreasing order of their efficiency. However, for very large item sets it is interesting to find s in linear time without the $O(n \log n)$ effort for sorting the items.

This can be done by an adaptation of the well-known linear time median algorithm due to Blum et al. [40] or the improved method by Dor and Zwick [112]. A detailed description of a linear median algorithm can be found e.g. in the book by Cormen et al. [92, Section 9.3].

Balas and Zemel [22] have been the first to use a linear median algorithm for finding the split item in linear time. Algorithm Split described in Figure 3.1 finds the split item in linear time and uses a linear median algorithm as a subprocedure. Its main idea is to find the value $r := \frac{p_s}{w_s}$. This is done by iteratively considering a candidate set S for the split item. In each iteration S is partitioned into three sets H, E and L indicating high, equal and low efficiencies. By computing the total weight for each of them we can decide in which of the sets the search for the item with efficiency r must be continued.

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

Algorithm Split:

$S := \{1, \ldots, n\}$ *set of candidates for s*
$\overline{w} := c$ *weight to be filled*
stop := false
repeat
  Compute r as median of the set $\{\frac{p_j}{w_j} \mid j \in S\}$
  r is a candidate for the efficiency of s
  $H := \{j \in S \mid \frac{p_j}{w_j} > r\}$ *items with efficiency greater than r*
  $E := \{j \in S \mid \frac{p_j}{w_j} = r\}$ *items with efficiency equal to r*
  $L := \{j \in S \mid \frac{p_j}{w_j} < r\}$ *items with efficiency smaller than r*
  if $\sum_{j \in H} w_j > \overline{w}$ then $S := H$ *r is too small*
  else if $\sum_{j \in H \cup E} w_j \leq \overline{w}$ then *r is too large*
    $S := L$, $\overline{w} := \overline{w} - \sum_{j \in H \cup E} w_j$
  else stop := true
  this means $\sum_{j \in H} w_j \leq \overline{w}$ *and* $\sum_{j \in H \cup E} w_j > \overline{w}$, *i.e.* $r = \frac{p_s}{w_s}$
until stop = true
$s := \min\{i \in E \mid \sum_{j \in E, j \leq i} w_j > \overline{w}\}$

Fig. 3.1. Split finds the split item s in linear time.

To show that the running time of split is $O(n)$ we note that every execution of the repeat-loop can be done in $O(|S|)$ time by the linear median algorithm mentioned above. Moreover, in every iteration $|S|$ is at most half as large as in the previous iteration by definition of the sets H and L through the median. Starting with $|S| = n$ the overall running time is bounded asymptotically by

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \leq 2n.$$

This can be summarized in the following statement.

Lemma 3.1.1 *The split item s of the knapsack problem can be computed in $O(n)$ time.*

3.2 Variable Reduction

It is clearly advantageous to reduce the size of a (KP) before it is solved. This holds in particular if the problem should be solved to optimality, but also heuristics and approximation algorithms will in general benefit from a smaller instance. Variable reduction considerably reduces the observed running time of algorithms for nearly all instances occurring in practice, although it does not decrease the theoretical worst-case running time as instances may be constructed where the reduction techniques have no effect.

Variable reduction algorithms may be seen as a special case of the branch-and-bound paradigm described in Section 2.4 where we only consider branching on a single variable at the root node. All variables are in turn chosen as the branching variable at the root node, and different branches are investigated corresponding to the domain of the variable. If a specific branch is inferior, then we may remove the corresponding value of the branching variable from the domain of the variable. The situation becomes particularly simple if the involved variables are binary as the elimination of one choice from the variable's domain immediately states the optimal value of the variable.

Considering (KP) we can branch on every variable x_j with $j = 1,\ldots,n$ yielding two subproblems as depicted in Figure 3.2.

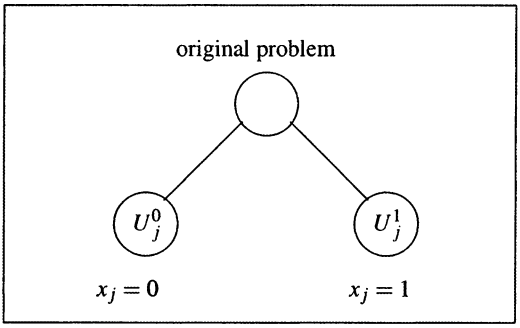


Fig. 3.2. Branching on the two possible solution values of x_j.

Let U_j^0 be an upper bound corresponding to the branch $x_j = 0$ and in a similar way U_j^1 an upper bound on the branch $x_j = 1$. Moreover, assume that an incumbent solution value z has been found in some way, e.g. by using the algorithm Greedy from Section 2.1. If $U_j^0 \leq z$, then we know that the branch $x_j = 0$ does not lead to an improved solution and thus we may fix the variable to $x_j = 1$. In a similar way $U_j^1 \leq z$ implies that $x_j = 0$ in every improved solution. All variables fixed at their optimal value may be removed from the problem thus decreasing the size of the instance.

As in all branch-and-bound algorithms the incumbent solution vector x corresponding to z should be saved. This is important as the reduction attempts to fix variables at their optimal value in every improved solution. If no improved solution is found, one should go back to the last incumbent solution.

As an upper bound in the reduction, any technique described in Section 2.4 can be used. In particular the bound U_{LP} from the linear programming relaxation described in Section 2.2 is fast to derive and thus well-suited for variable reduction.

3.3 Storage Reduction in Dynamic Programming

Since dynamic programming is a general tool not only to compute optimal solutions of the knapsack problem or its variants and extensions but also for the construction of approximation algorithms (see Section 2.6), we will introduce a general procedure to reduce the space requirements of dynamic programming. This procedure can be used for most of the dynamic programming schemes described in this book.

The principle of this method originates in the context of the knapsack family and was first developed by Kellerer et al. [268] in a very special and more complicated version for the approximation of the subset sum problem. It was adapted by Kellerer and Pferschy [267] in a different way for the approximation of the knapsack problem (see Section 6.2) and presented in a simplified and generalized form in Pferschy [374]. Independently, Hirschberg [229] used a similar method in the context of computing maximal common subsequences.

The method can be useful in general for problems where it takes more effort to compute the actual optimal solution set than the optimal solution value. By a recursive *divide and conquer* strategy the problem is broken down into smaller and smaller subproblems until it is easy to find the optimal solution sets of these small subproblems, which are then combined to an overall solution. Under certain conditions this recursion can be performed in the same running time as required to compute only the optimal solution value.

The setup for our description of the new algorithmic framework Recursive-DP follows the procedure DP-2 from Section 2.3. We are given a set N of input elements (corresponding to the items) and an integer parameter C (corresponding to the capacity). A combinatorial optimization problem $P(N,C)$ is defined with optimal solution set $X^*(N,C)$ and optimal solution value $z^*(N,C)$. The recursive algorithm in its general form is relatively simple and will be described in Figure 3.3. It is started by setting $N^* := \emptyset$ and $C^* := 0$ and performing Recursive-DP(N,C). At the end of the recursion the optimal solution set can be finally found as $X^*(N,C) := X^*(N^*,C^*)$.

Note that the application of this algorithmic framework to a particular problem leaves plenty of room for improvements. Some ideas to this point are outlined at the end of this section.

Recursive-DP can be applied successfully if the following conditions are fulfilled by the given problem:

(1) There is a procedure Solve(N,C) which computes for every integer $c = 0,\ldots,C$ the value $z^*(N,c)$ and requires $O(|N| \cdot C)$ time and $O(|N| + C)$ space.

(2) If $|N| = 1$, then Solve(N,C) also computes $X^*(N,c)$ for every $c = 0,\ldots,C$.

(3) For every partitioning N_1,N_2 of N there exist C_1,C_2 such that $C_1 + C_2 = C$ and $z^*(N_1,C_1) + z^*(N_2,C_2) = z^*(N,C)$.

(4) For every partitioning N_1, N_2 of N with $|N_1| = 1$ and arbitrary parameters C_1, C_2 there exists a procedure Merge which combines $X^*(N_1, C_1)$ and $X^*(N_2, C_2)$ into $X^*(N_1 \cup N_2, C_1 + C_2)$ in $O(C_1)$ time.

Algorithm Recursive-DP(N,C):
$X^*(N^*,C^*)$ *is the currently known overall solution set.*

Divide
Partition N into two disjoint subsets N_1 and N_2 with $|N_1| \approx |N_2|$.
Perform Solve(N_1,C) returning $z^*(N_1,c)$ for $c = 0,\ldots,C$.
Perform Solve(N_2,C) returning $z^*(N_2,c)$ for $c = 0,\ldots,C$.
Find C_1, C_2 with $C_1 + C_2 = C$ and $z^*(N_1,C_1) + z^*(N_2,C_2) = z^*(N,C)$.

Conquer
if $|N_1| = 1$ then
    Merge $X^*(N_1,C_1)$ and $X^*(N^*,C^*)$ into $X^*(N_1 \cup N^*, C_1 + C^*)$
    $N^* := N^* \cup N_1$, $C^* := C^* + C_1$
if $|N_2| = 1$ then
    Merge $X^*(N_2,C_2)$ and $X^*(N^*,C^*)$ into $X^*(N_2 \cup N^*, C_2 + C^*)$
    $N^* := N^* \cup N_2$, $C^* := C^* + C_2$

Recursion
if $|N_1| > 1$ then perform Recursive-DP(N_1,C_1)
if $|N_2| > 1$ then perform Recursive-DP(N_2,C_2)

Fig. 3.3. Recursive-DP computing the optimal solution set by divide and conquer.

Correctness:

To show that performing Recursive-DP(N,C) computes the correct optimal solution set $X^*(N,C)$ we will use the imposed conditions.

First of all condition (1) delivers procedure Solve as required in the **divide** part. The existence of the required parameters C_1 and C_2 at the end of the **divide** part is guaranteed by condition (3).

It can be easily seen that for every single element $j \in N$ there is some point in the recursive partitioning of N induced by Recursive-DP(N,C) where one of the two subsets N_1, N_2 generated in the **divide** part contains only this element j. However, if $|N_1| = 1$ or $|N_2| = 1$ then according to (2) also the optimal solution set of the corresponding subproblem can be computed and combined with, respectively added to the existing partial solution $X^*(N^*,C^*)$ through procedure Merge as given by condition (4). Following the successive partitioning according to (3) and applying this combination mechanism in the **conquer** part for all subproblems consisting of single elements finally an overall optimal solution set $X^*(N^*,C^*)$ is collected.

Running Time:

The main computational effort of an execution of Recursive-DP(N',C') (without the recursive calls) is given by the two performances of Solve and requires $O(|N'| \cdot C')$ time because of condition (1). Determining C_1 and C_2 at the end of the **divide** part can be done in $O(C')$ time by going through the arrays $z^*(N_1,\cdot)$ and $z^*(N_2,\cdot)$ in opposite directions and searching for a combination of capacities summing up to C' and yielding the highest total value. The combination of the solution set in the **conquer** part by procedure Merge (if it is performed at all) requires at most $O(C')$ time due to condition (4).

To analyze the overall running time, a representation of the recursive structure of Recursive-DP as a binary rooted tree is used. Each node in the tree corresponds to a call to Recursive-DP with Recursive-DP(N,C) corresponding to the root node.

A node is said to have *level* ℓ in the tree if there are $\ell - 1$ nodes on the path to the root node, which has level 0. Obviously, the level of a node is equivalent to its recursion depth and gives the number of bipartitionings of the initial set N. Hence, the cardinality of the set of elements in a node of level ℓ is bounded by $|N|/2^\ell$ and the maximum level of a node is $\lceil \log |N| \rceil - 1$ because a node with at most 2 elements is a leaf of the tree.

Let C_j^ℓ, $j = 1,\ldots,2^\ell$, denote the (capacity) parameter of node j in level ℓ. Note that by construction of C_1 and C_2 we have $\sum_{j=1}^{2^\ell} C_j^\ell = C$ for every level ℓ. The running time for all nodes in level ℓ is given in complexity by

$$\sum_{j=1}^{2^\ell} \frac{|N|}{2^\ell} \cdot C_j^\ell = \frac{|N|}{2^\ell} \cdot C.$$

Summing up over all levels we get a running time complexity of

$$\sum_{\ell=0}^{\log |N|} \frac{|N|}{2^\ell} \cdot C \leq |N| \cdot C \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) = 2|N| \cdot C. \tag{3.1}$$

Space Requirements:

In every execution of Recursive-DP(N',C') we have to compute and store the results of Solve, i.e. the arrays $z^*(N_1,\cdot)$ and $z^*(N_2,\cdot)$, which takes $O(|N'| + C')$ space by condition (1). However, this information is only required to compute the two parameters C_1 and C_2 and to perform **Conquer**. Hence, it can be deleted before going into the **Recursion**. The remaining operations clearly do not require any relevant storage.

Throughout the execution of the recursive structure we only need $O(|N'| + C')$ space for the current performance of procedure Recursive-DP(N',C') and we have to

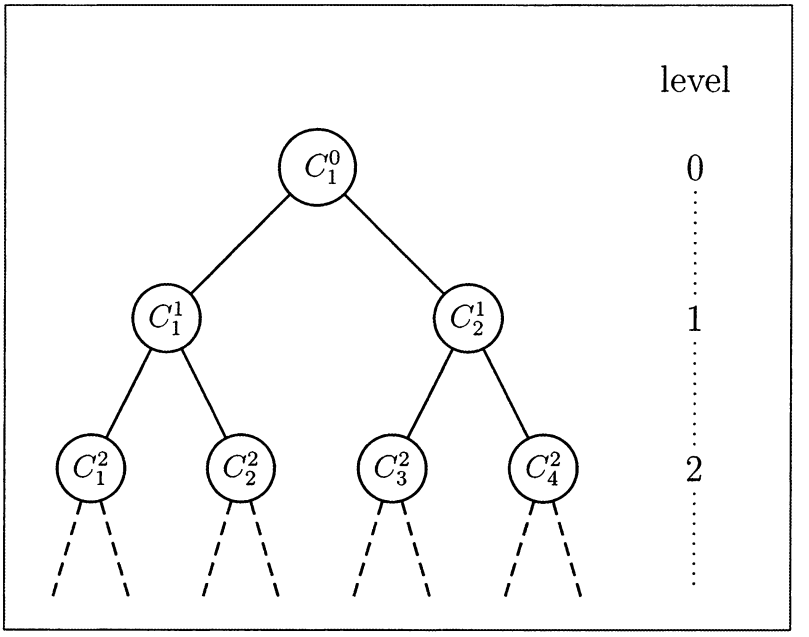


Fig. 3.4. A binary rooted tree representing the recursive structure of Recursive-DP by nodes labeled with the capacity parameters.

store $X^*(N^*,C^*)$ and keep track of the subdivision of N. This can be done by standard methods without increasing the overall space complexity of $O(|N|+C)$.

Summarizing, we conclude:

Theorem 3.3.1 *If conditions (1)–(4) are fulfilled, then the optimal solution set of $P(N,C)$ can be computed by algorithm* Recursive-DP(N,C) *using $O(|N|\cdot C)$ time and $O(|N|+C)$ space.*

As an immediate application we state the improved complexity of a dynamic programming algorithm for the knapsack problem. All other applications will be addressed in the corresponding sections.

Corollary 3.3.2 *The knapsack problem (KP) can be solved in $O(nc)$ time and $O(n+c)$ space.*

Proof. Setting $N = \{1,\ldots,n\}$ as set of items and $C = c$ as the capacity, dynamic programming by weights without storing the solution sets (i.e. algorithm DP-2 from Section 2.3) satisfies (1). Condition (2) is trivial. To find C_1 and C_2 fulfilling (3) the two dynamic programming arrays have to be scanned as described above to find a

combination of capacities summing up to c with maximal total profit. A simple set union operation delivers procedure Merge in (4) and can be done even in constant time by appropriate pointer techniques. □

Of course Recursive-DP can be improved and fine tuned if it is applied to a particular problem. Moreover, the four required conditions can be softened in some way. For instance, condition (2) can be often adapted in the sense that also for some moderate size of $|N|$ the optimal solution set is known. In other situations a small part of the solution set may be available after every execution of Solve such that N_1 and N_2 can be reduced before performing the recursive calls. Note that the analysis does not change even if Recursive-DP(N',C') requires $O(|N'| \cdot C')$ time to combine partial solutions.

Note that Magazine and Oguz [314] gave a related approach based on the successive bipartitioning of the item set for the approximation of the knapsack problem which also reduces space but at the cost of increasing time.

The *separability property* of the knapsack problem was discussed earlier by Horowitz and Sahni [238] in a way related to condition (3). They used this property to solve (KP) in $O(\sqrt{2^n})$ worst-case time. The idea of the *Horowitz and Sahni decomposition*, which can be seen as a distant predecessor to Recursive-DP, is to divide the problem into two equally sized parts, and enumerate each subproblem through dynamic programming. The two tables of optimal profit sums for each possible weight sum are assumed to be sorted according to increasing weights. To obtain a global optimum, the two sets are merged by considering pairs of states which have the largest possible weight sum not exceeding c. This may be done in time proportional to the size of the sets as both sets are sorted. This technique gives an improvement over a complete enumeration by a factor of a square-root. It can also be used for a parallel algorithm for (KP).

3.4 Dynamic Programming with Lists

The simple dynamic programming algorithm DP-1 for (KP) of Section 2.3 evaluated a table of entries $z_j(d)$ for $j = 0,\ldots,n$ and $d = 0,\ldots,c$ using $O(nc)$ time. Although the following technique does not change the worst-case complexity, it may considerably improve the computation time for practical instances. The idea is to consider each pair $(d,z_j(d))$ of the dynamic programming array as a *state* $(\overline{w},\overline{p})$. Thus each state is a pair of two integers, where $\overline{w}$ denotes a given capacity, and $\overline{p}$ denotes the maximum profit sum obtainable for this capacity when considering the subproblem defined on the first j items.

For each value of $j = 1,\ldots,n$ we write the states as a list

$$L_j = \langle (\overline{w}_{1j},\overline{p}_{1j}),(\overline{w}_{2j},\overline{p}_{2j}),\ldots,(\overline{w}_{mj},\overline{p}_{mj})\rangle \tag{3.2}$$

where the number of states is in $O(c)$ as the weight sums $\overline{w}$ must be integers between 0 and c. A list may be pruned by using the concept of *dominance*.

Definition 3.4.1 *Assume that* $(\overline{w},\overline{p})$ *and* $(\overline{w}',\overline{p}')$ *are two states in a list* L_j. *If* $\overline{w} < \overline{w}'$ *and* $\overline{p} \geq \overline{p}'$ *or if* $\overline{w} \leq \overline{w}'$ *and* $\overline{p} > \overline{p}'$, *then we say that the first states* dominates *the second state.*

For the unbounded knapsack problem (UKP) we will introduce considerable extensions of this notion of dominance in Section 8.2. From the optimal substructure property of (KP) (see Section 2.3) we immediately get the following.

Corollary 3.4.2 *A dominated state* $(\overline{w}',\overline{p}')$ *will never result in an optimal solution, and thus it may be removed from the list* L_j.

If we change the dynamic programming recursion into a list representation, we may hope that a majority of the states will be deleted due to dominance, thus improving the practical running time.

For $j = 0$ we have the list $L_0 = \langle(0,0)\rangle$ as the only undominated state with weight and profit sum equal to zero. A subsequent list L_j is obtained from the list L_{j-1} in two steps. At first the weight w_j and profit p_j of item j is added to the weights and profits of all states contained in list L_{j-1} producing a new list L'_{j-1}. This componentwise addition will be denoted by

$$L'_{j-1} := L_{j-1} \oplus (w_j, p_j). \tag{3.3}$$

In the second step the two lists L_{j-1} and L'_{j-1} are *merged* to obtain L_j. The principal procedure is outlined in the algorithm DP-with-Lists depicted in Figure 3.5.

Algorithm DP-with-Lists:
$L_0 := \langle(0,0)\rangle$
for $j := 1$ **to** n
  $L'_{j-1} := L_{j-1} \oplus (w_j, p_j)$ *add* (w_j, p_j) *to all elements in* L_{j-1}
  delete all states $(\overline{w},\overline{p}) \in L'_{j-1}$ with $\overline{w} > c$
  $L_j :=$ MergeLists(L_{j-1}, L'_{j-1})
return the largest state in L_n

Fig. 3.5. DP-with-Lists using componentwise addition to obtain states of profit and weight.

During DP-with-Lists we keep the lists L_j sorted according to increasing weights and profits, such that the merging of lists can be performed efficiently. This is done in subprocedure Merge-Lists. Note that any set of non-dominated states is always totally ordered, i.e. the sorting of states e.g. by weights automatically leads to an

ordering also by profits. Hence, we can refer to the *smallest* and *largest* state of every list corresponding to the state with the smallest, respectively largest weight and the *last* state in the list is always identical to the largest state.

Given two lists L, L' to be merged into a list L'', we denote the largest state in L'' by $(\overline{w}'', \overline{p}'')$. We repeatedly remove the smallest state $(\overline{w}, \overline{p})$ of both lists L and L' and insert it into list L'' if the state is not dominated and if it is not equal to $(\overline{w}'', \overline{p}'')$. Since we consider the states in increasing order of weight sum, we have $\overline{w} \geq \overline{w}''$. This results in one of the following situations:

- If $\overline{p} \leq \overline{p}''$, then $(\overline{w}, \overline{p})$ is dominated by $(\overline{w}'', \overline{p}'')$ or $(\overline{w}, \overline{p}) = (\overline{w}'', \overline{p}'')$. Thus, we may surpass the former.
- If $\overline{p} > \overline{p}''$ and $\overline{w} = \overline{w}''$ then $(\overline{w}'', \overline{p}'')$ is dominated by $(\overline{w}, \overline{p})$ and we replace the former state by the latter state in L''.
- If $\overline{p} > \overline{p}''$ and $\overline{w} > \overline{w}''$ there is no dominance between the two states, and thus we add state $(\overline{w}, \overline{p})$ to L''.

This leads to the algorithm outlined in Figure 3.6.

```
Procedure Merge-Lists(L, L'):
L, L' and L'' are lists of states sorted in increasing order of profits
and weights
L'' is the result list attained by merging L and L'

add the state (∞, 0) to the end of both lists L, L'.
L'' := ⟨(∞, 0)⟩
repeat
    choose the state (w̄, p̄) with smallest weight in L and L' and
    delete it from the corresponding list.
    if w̄ ≠ ∞ then
        assume that (w̄'', p̄'') is the largest state in L''
        if (p̄ > p̄'') then
            if (w̄ ≤ w̄'') then delete (w̄'', p̄'') from L''
            add (w̄, p̄) to the end of list L''
until w̄ = ∞
return L''
```

Fig. 3.6. Merging of lists in increasing order of profits and weights.

The running time of Merge-Lists is $O(c)$ since both lists L and L' have length at most $c+1$ and all operations within the central repeat loop can be performed in constant time by keeping pointers to the first and last state, respectively. This implies that the running time of DP-with-Lists becomes $O(nc)$, i.e. the same complexity as in DP-1. But whereas the worst-case and best-case solution time of DP-1 are the same, the best-case solution time of DP-with-Lists may be as low as $O(n)$ if the first

item considered has a very large profit, and the weight equals c. Empirical results show that in practical applications a large number of states may be deleted due to dominance, hence making the present version preferable.

The following property of DP-with-Lists follows immediately from its definition. It is contributed to a classical paper by Nemhauser and Ullmann [358].

Lemma 3.4.3 *If the total number of undominated states of an instance of (KP) is bounded by B, then (KP) can be solved by* DP-with-Lists *in $O(nB)$ time.*

Example: (cont.) We consider again our example from Section 2.1. Then Figure 3.7 shows the undominated states obtained by algorithm DP-with-Lists. j denotes the current item considered and k is a counter for the different number of states. □

$k \backslash j$	0	1	2	3	4	5	6	7
1	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)
2		(6,2)	(6,2)	(6,2)	(6,2)	(6,2)	(6,2)	(6,2)
3			(11,5)	(11,5)	(11,5)	(11,5)	(11,5)	(11,5)
4				(14,8)	(14,8)	(12,7)	(12,7)	(12,7)
5					(15,9)	(14,8)	(14,8)	(14,8)
6						(15,9)	(15,9)	(15,9)

Fig. 3.7. The different states of our example returned by algorithm DP-with-Lists.

3.5 Combining Dynamic Programming and Upper Bounds

In Section 2.3 it was established that a problem can be solved through dynamic programming if it has the property of optimal substructure. If it is possible to derive also upper bounds on every solution value of a subproblem, then the dynamic programming algorithm may be combined with elements of a branch-and-bound algorithm (see e.g. Morin and Marsten [350]). The benefits of such an approach are obvious: The algorithm will still have the worst-case complexity of the dynamic programming algorithm while in the best-case it may exploit upper bounds and prune the search considerably as in branch-and-bound.

To understand the idea, one must remember from Section 3.4 that a state $(\overline{w},\overline{p}) = (d, z_j(d))$ in dynamic programming corresponds to an optimal solution of a subproblem with capacity d and items $1,\ldots,j$ of the original problem. But a state can also be seen as a node in the branch-and-bound tree where we have fixed the values of variables x_i for $i = 1,\ldots,j$ such that the corresponding weight sum is $\overline{w} = \sum_{i=1}^{j} w_i x_i$ and profit sum is $\overline{p} = \sum_{i=1}^{j} p_i x_i$. Thus both approaches are based on an enumeration

of the variables, where dynamic programming is using dominance to improve the complexity, while branch-and-bound is using bounding to prune the search.

In order to combine elements of the two approaches it is first necessary to convert the dynamic programming recursion to DP-with-Lists as described in Section 3.4. Then, a bounding function $U(\overline{w},\overline{p})$ must be chosen which derives an upper bound for every state $(\overline{w},\overline{p})$. As in Section 2.4 we fathom a state $(\overline{w},\overline{p})$ if the corresponding upper bound $U(\overline{w},\overline{p})$ does not exceed the incumbent solution z.

To illustrate the principle for (KP) we use DP-with-Lists as introduced in Section 3.4. As an upper bound on a state we choose the LP-relaxation (LKP) defined in (2.3). Recall that the state $(\overline{w},\overline{p})$ corresponds to an optimal solution to problem $(\mathrm{KP}_j(d))$ defined in (2.7). Hence, $\overline{p}$ is the optimal solution for

$$\begin{aligned} \text{maximize} \quad & \sum_{\ell=1}^{j} p_\ell x_\ell \\ \text{subject to} \quad & \sum_{\ell=1}^{j} w_\ell x_\ell \leq d, \\ & x_\ell \in \{0,1\}, \quad \ell = 1,\ldots,j. \end{aligned}$$

An upper bound on the state $(\overline{w},\overline{p})$ is given by

$$\begin{aligned} U(\overline{p},\overline{w}) = \overline{p} \; + \; \text{maximize} \quad & \sum_{\ell=j+1}^{n} p_\ell x_\ell \\ \text{subject to} \quad & \sum_{\ell=j+1}^{n} w_\ell x_\ell \leq c - \overline{w} \\ & 0 \leq x_\ell \leq 1, \quad \ell = j+1,\ldots,n. \end{aligned}$$

This leads to the algorithm DP-with-Upper-Bounds described in Figure 3.8:

As the bounds are evaluated in $O(n)$ time, the running time of this algorithm becomes $O(n^2c)$. The worst-case complexity is however not an appropriate measure for the practical performance of this algorithm since the pruning of the states due to the upper bounds may significantly decrease the number of states in each list and we may also use upper bounds which can be computed in constant time thus getting a running time of $O(nc)$.

3.6 Balancing

A different technique to limit the search of dynamic programming is *balancing*. A *balanced solution* is loosely speaking a solution which is sufficiently filled, i.e. the

Algorithm DP-with-Upper-Bounds:
  let z be a heuristic solution
  $L_0 := \langle (0,0) \rangle$
  for $j := 1$ **to** n
    $L'_{j-1} := L_{j-1} \oplus (w_j, p_j)$
    delete all states $(\overline{w}, \overline{p}) \in L'_{j-1}$ with $\overline{w} > c$
    $L_j :=$ Merge-Lists(L_{j-1}, L'_{j-1})
    compute an upper bound $U(\overline{w}, \overline{p})$
    delete all states $(\overline{w}, \overline{p}) \in L_j$ with $U(\overline{w}, \overline{p}) \leq z$.
    let $(\overline{w}, \overline{p})$ be the largest state in L_j
    set $z := \max\{z, \overline{p}\}$
  return z

Fig. 3.8. DP-with-Upper-Bounds improving DP-with-Lists by using upper bounds on the states.

weightsum does not differ from the capacity by more than the weight of a single item. An optimal solution is balanced, and hence the dynamic programming recursion may be limited to consider only balanced states. Pisinger [387] showed that using balancing, several problems from the knapsack family are solvable in linear time provided that the weights w_j and profits p_j are bounded by a constant. For the subset sum problem one gets the attractive solution time of $O(n w_{\max})$ which dominates the running time $O(nc)$ of DP-1. For knapsack problems balancing yields a running time complexity of $O(n p_{\max} w_{\max})$ which may be attractive if the profits and weights of the items are not too large.

Assume the items to be sorted in decreasing order of their efficiencies according to (2.2). To be more formal a *balanced filling* is a solution x obtained from the split solution $\hat{x}$ (defined in Section 2.2) through *balanced operations* as follows:

- The split solution $\hat{x}$ is a balanced filling.
- *Balanced insert:* If we have a balanced filling x with $\sum_{j=1}^{n} w_j x_j \leq c$ and change a variable x_b, $b \geq s$, from $x_b = 0$ to $x_b = 1$, then the new filling is also balanced.
- *Balanced remove:* If we have a balanced filling x with $\sum_{j=1}^{n} w_j x_j > c$ and change a variable x_a, $a < s$, from $x_a = 1$ to $x_a = 0$, then the new filling is also balanced.

The relevance of balanced fillings is established in the following theorem due to Pisinger [387].

Theorem 3.6.1 *An optimal solution to (KP) is a balanced filling, i.e. it may be obtained from the split solution through balanced operations.*

Proof. Assume that the optimal solution is given by x^*. Let $a_1, \ldots, a_\alpha$ be the indices $a_i < s$ where $x^*_{a_i} = 0$, and $b_1, \ldots, b_\beta$ be the indices $b_i \geq s$ where $x^*_{b_i} = 1$. Order the indices such that

$$a_\alpha < \cdots < a_1 < s \leq b_1 < \cdots < b_\beta.$$

Starting from the split solution $x = \hat{x}$ we perform balanced operations in order to reach x^*. As the split solution satisfies $\sum_{j=1}^{n} w_j x_j \leq c$, we must insert an item b_1, thus set $x_{b_1} = 1$. If the hereby obtained weight sum $\sum_{j=1}^{n} w_j x_j$ is greater than c, we remove item a_1 by setting $x_{a_1} = 0$, otherwise we insert the next item b_2. Continuing in this way, finally one of the following three situations will occur:

1. All the changes corresponding to $\{a_1, \ldots, a_\alpha\}$ and $\{b_1, \ldots, b_\beta\}$ have been made, meaning that we have reached the optimal solution x^* through balanced operations. This is the desired situation.
2. We reach a situation where $\sum_{j=1}^{n} w_j x_j > c$ and all indices a_i have been used but some b_i have not been used. This however implies that the corresponding solution is a subset of the optimal solution set. Consequently, x^* could not be a feasible solution from the start as the knapsack is filled and we still have to insert items.
3. A similar situation occurs when $\sum_{j=1}^{n} w_j x_j \leq c$ is reached and all indices b_i have been used, but some a_i are missing. This implies that x^* cannot be an optimal solution, as a better feasible solution can be obtained by *not* removing the remaining items a_i. □

From the proof of Theorem 3.6.1 we immediately get the following two observations:

Observation 3.6.2 *An optimal solution to (KP) can be obtained by performing balanced operations on the split solution $\hat{x}$ such that items $b \geq s$ are inserted in increasing order of indices, and items $a < s$ are removed in decreasing order of indices.*

Observation 3.6.3 *All balanced solutions x satisfy that*

$$c - w_{\max} < \sum_{j=1}^{n} w_j x_j \leq c + w_{\max}.$$

As an illustration we will apply the concept of balancing to a dynamic programming algorithm for the *subset sum problem* (SSP) defined in the beginning of Section 1.2. Since (SSP) has the simplest structure of all problems in the knapsack family it should give the best illustration of a non-trivial balancing algorithm.

For this purpose we define a table $g(b, \overline{w})$ for $b = s, \ldots, n$ and $c - w_{\max} < \overline{w} \leq c + w_{\max}$ as follows.

$$g(b,\overline{w}) := \max_{a=1,\ldots,b+1} \left\{ a \;\middle|\; \begin{array}{l} \text{a balanced filling } x \text{ exists with} \\ \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{b} w_j x_j = \overline{w},\ x_j \in \{0,1\},\ j = a, \ldots, b \end{array} \right\} \tag{3.4}$$

An entry $g(b,\overline{w}) = a$ means that there exists a balanced solution with value $\overline{w}$ containing the item set $\{1,\ldots,a-1\}$ and a subset of items $\{a+1,\ldots,b\}$ but not item a. Furthermore, it does not contain any items with index larger than b. If no balanced filling exists we set $g(b,\overline{w}) := 0$. Note, that we may assume

$$g(b,\overline{w}) \geq 2 \quad \text{for} \quad \overline{w} > c,\ b = s,\ldots,n, \tag{3.5}$$

since if $g(b,\overline{w}) = 1$ the corresponding balanced solution will not lead to a feasible solution, as we cannot remove any more items when $a = 1$.

Assuming that we somehow had calculated the table, then we could solve (SSP) by choosing z^* as the maximum $\overline{w} \leq c$ such that $g(n,\overline{w}) \neq 0$.

The table $g(b,\overline{w})$ is calculated in a non-trivial way in order to obtain a tight bound on the running time. Thus the algorithmic description will be split into a number of phases. First, we will describe how the table is initialized. Then we show an enumeration algorithm which runs through all balanced solutions and which can be used to fill table $g(b,\overline{w})$ in a straightforward way. In order to improve the complexity of the recursive procedure we store intermediate results and use dominance to avoid repeated or unfruitful work. Finally, we will prove the time and space complexity of the algorithm.

a) Initialization of the table. Before starting the recursion we may initialize table $g(b,\overline{w})$, mainly for technical reasons, as follows for $b = s,\ldots,n$:

$$g(b,\overline{w}) := 0 \quad \text{for} \quad c - w_{\max} < \overline{w} \leq c, \tag{3.6}$$

$$g(b,\overline{w}) := 1 \quad \text{for} \quad c < \overline{w} \leq c + w_{\max} \tag{3.7}$$

For $\overline{w} \leq c$ the initialization means that no balanced solutions exist. The initialization of $g(b,\overline{w}) = 1$ for $\overline{w} > c$ is going to be used in Step b). It is worth noting that we do not affect the optimal solution by this initialization due to (3.5).

b) Enumerating all balanced solutions. Table $g(b,\overline{w})$ can easily be filled by considering all balanced solutions and for each considered solution checking whether $g(b,\overline{w})$ can be improved. This can be done by the algorithm Balanced-Search outlined in Figure 3.9. The algorithm runs through the complete *tree of balanced solutions*, where each node corresponds to one balanced solution and hence is characterized by the triple $(a,b,\overline{w})$ while the edges of this tree are defined implicitly later on. The enumeration is based on a *pool of open nodes*, where in each iteration one open node is selected, a branching occurs and possible new "children nodes" are added to the pool.

Initially the pool P has only one node $(a,b,\overline{w}) = (s,s-1,\hat{w})$, as the split solution is the first balanced filling.

In lines 2-3 we repeatedly choose an open node, until the pool of nodes is empty. We may initially assume that the nodes are taken in arbitrary order, although a different strategy will be described in the sequel.

Algorithm Balanced-Search:

```
1  initialize the pool of open nodes P := {(s, s − 1, ŵ)}
2  while P ≠ ∅
3      choose an open node (a, b, w̄) from P and remove it from P
4      g(b, w̄) := max{g(b, w̄), a}
5      if w̄ ≤ c then
6          if b ≤ n then
7              P := P ∪ {(a, b + 1, w̄)}
8              P := P ∪ {(a, b + 1, w̄ + w_b)}
9      else
10         for j := a − 1 downto g(b − 1, w̄) do
11             P := P ∪ {(j, b, w̄ − w_j)}
```

Fig. 3.9. Balanced-Search using a pool P of open nodes.

Line 4 updates the table $g(b,\overline{w})$ such that it contains the maximum value of a which makes it possible to obtain the weight sum $\overline{w}$ by only considering balanced insertions and removals of items $a,\ldots,b$.

Lines 5–11 branch on the current node, generating new children nodes. If the current weight sum $\overline{w}$ is not larger than c then we should insert an item in lines 6–8. Actually the two lines express the dichotomy that either we may choose item b or we may omit it. Obviously, we only consider item b if $b \leq n$.

Lines 10–11 consider the case where the current weight sum $\overline{w}$ exceeds the capacity c, and thus we should remove an item. This is done by choosing an item $j < a$ which is removed. Actually line 11 should consider all values of j between $a-1$ and down to 1. But if $a' = g(b-1,\overline{w})$ is larger than 1, then it means that at some earlier point of the algorithm we considered a node $(a',b-1,\overline{w})$ in P. At that moment we had investigated the removal of all items $j < a'$, so we do not have to repeat the work.

c) Dominance of nodes. We may apply dominance to remove some of the open nodes. Assume that two open nodes $(a,b,\overline{w})$ and $(a',b',\overline{w}')$ are in the pool P at the same time. If

$$b = b', \overline{w} = \overline{w}' \text{ and } a \geq a', \tag{3.8}$$

then the first node dominates the latter, and we may remove the latter. This is due to the fact that if we can obtain a balanced filling with weight sum $\overline{w}$ by looking only on items $\{a,\ldots,b\} \subseteq \{a',\ldots,b'\}$, then we can also obtain all subsequent balanced fillings obtainable from $(a',b',\overline{w}')$ by considering $(a,b,\overline{w})$ instead.

d) Evaluation order of nodes. Due to the concept of dominance we will only have one open node for each value of b and $\overline{w}$. This means that we can store the open nodes as triples $(a,b,\overline{w})$ in a table $P(b,\overline{w})$ of dimensions $b = s,\ldots,n$ and $c - w_{\max} < \overline{w} \leq c + w_{\max}$. Notice, that P has the same size as table g. The test of dominance can be performed in constant time each time we save a node $(a,b,\overline{w})$ to the pool P, as we only need a simple look-up in table $P(b,\overline{w})$.

We now choose the order of evaluation for the open nodes such that the problems with smallest value of b are considered first. If more subproblems have the same value of b then we choose the subproblem with largest value of $\overline{w}$.

Each open node in Balanced-Search either will increase the value of b, or decrease the value of $\overline{w}$ with an unchanged b. Thus with respect to the chosen order of evaluation, all new subproblems will be added after the currently investigated node in table P. This means that we only need to run through the table of open subproblems once, namely for increasing values of b and decreasing values of $\overline{w}$. Hence line 3 of algorithm Balanced-Search simply runs through table P for increasing values of b and decreasing values of $\overline{w}$. It also follows that the number of nodes considered in line 2 of Balanced-Search is bounded by $O(n\,w_{\max})$.

To speed up the algorithm, we copy table $g(b,\overline{w})$ to positions $g(b+1,\overline{w})$ each time we increase b in the outer loop. This is obviously allowed, since if a balanced solution with weight sum $\overline{w}$ can be found on items $a,\ldots,b$ then obviously a balanced solution with the same weight sum $\overline{w}$ can be found on items $a,\ldots,b+1$. As will be clear from the following, this improves the time complexity as the loop in line 10 of Balanced-Search typically will terminate earlier.

e) Time and space complexity. The space complexity is easy to derive, since our table $g(b,\overline{w})$ has dimension $n \cdot 2w_{\max}$ and the table of open nodes has the same dimension. Thus the space complexity is $O(n\,w_{\max})$.

The initialization of the two tables takes $O(n\,w_{\max})$ time. Since we noticed above that the outer loop of algorithm Balanced-Search will consider at most $O(n\,w_{\max})$ nodes, lines 4–8 will be evaluated the same number of times. Evaluation of lines 10–11 cannot be bounded in the same simple way, as the loop may run over several values of j. One should however notice that for a given value of $\overline{w}$ and a given value of b, line 10 is evaluated for $j = g(b,\overline{w}) - 1$ downto $g(b-1,\overline{w})$, i.e. in total

$$g(b,\overline{w}) - g(b-1,\overline{w}) \tag{3.9}$$

times. If we sum up for a given value of $\overline{w}$ the computational effort for all values of b, we get the time bound

$$\sum_{b=s+1}^{n} \Big(g(b,\overline{w}) - g(b-1,\overline{w})\Big) = g(n,\overline{w}) - g(s,\overline{w}) \leq n \tag{3.10}$$

since $g(n,\overline{w}) \leq n$ and $g(s,\overline{w}) = 1$. Thus the computational effort of lines 10–11 taken over all values of $\overline{w}$ and b will also be limited by $O(n\,w_{\max})$. This proves the overall time complexity of $O(n\,w_{\max})$.

f) Table of open subproblems The pool P of open nodes is actually obsolete, since we may save the open nodes in table $g(b,\overline{w})$. Each value of $a = g(b,\overline{w}) > 0$ corresponds to an open node $(a,b,\overline{w})$ in P. It is not necessary to distinguish between open nodes and already treated nodes since we have a fixed order of running through the table $g(b,\overline{w})$ as described in Step d).

The final implementation of the algorithm will be described in more details in Section 4.1.

3.7 Word RAM Algorithms

Algorithms exploiting *word-parallelism* (i.e. parallelism at bit level) have been studied during the last decade in theoretical computer science. For an excellent survey on sorting and searching algorithms on the word RAM we refer to Hagerup [205]. These algorithms have improved the time complexity of several problems and given a new insight into the design of algorithms. A natural extension is to try to use the same principle for solving $\mathcal{NP}$-hard problems through dynamic programming as proposed by Pisinger [392].

We will use the *word RAM model of computation* [151] (see also Cormen et al. [92, Section 2.2]), meaning that we can e.g. perform binary `and`, binary `or`, and bitwise `shift` with up to W bits in constant time on words of size W. By storing subsolutions as bits in a word in the dynamic programming algorithm, we may work on W subsolutions in a single operation hence in the best case getting a speed-up of a factor W in the computational time. In the following we will assume that all bits in a word are numbered from left to right $0, 1, \ldots, W-1$ and words are numbered in a similar way. Figure 3.10 shows the words and bits for a word size of $W = 8$.

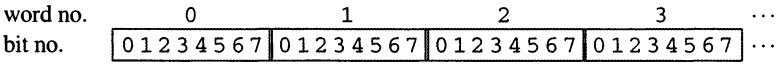


Fig. 3.10. Numbering of words and bits

We will illustrate the word RAM technique on the *subset sum problem* (SSP) defined in the beginning of Section 1.2. It is natural to assume that the word size W of the computer is sufficiently large to hold all coefficients in the input, hence $W \geq \log c$. (Note that the logarithm is base 2.) If this was not the case, we had to change the implicit assumption that we may perform arithmetic operations on the coefficients in constant time. Sometimes we will use the stronger assumption that the word size W corresponds to the size largest coefficients in the input data, i.e.

$$W \text{ is in } \Theta(\log c). \tag{3.11}$$

With $W \geq \log c$ we will show how to decrease the running time of the Bellman recursion (2.8) from $O(nc)$ to $O(nc/\log c)$ by using word parallelism. We use the following invariant defined for $j = 0, \ldots, n$ and $\overline{w} = 0, \ldots, c$:

$$\begin{aligned} &g(j, \overline{w}) = 1 \Leftrightarrow \\ &\text{there is a subset of } \{w_1, \ldots, w_j\} \text{ with overall sum } \overline{w} \end{aligned} \tag{3.12}$$

		$\overline{w}$ (capacity)									
$z_j(\overline{w})$		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	3	3	3	3	3	3	3
j	2	0	0	0	3	3	5	5	5	8	8
	3	0	1	1	3	4	5	6	6	8	9
	4	0	1	1	3	4	5	6	7	8	9

		$\overline{w}$ (capacity)									
$g(j,\overline{w})$		0	1	2	3	4	5	6	7	8	9
	0	1	0	0	0	0	0	0	0	0	0
	1	1	0	0	1	0	0	0	0	0	0
j	2	1	0	0	1	0	1	0	0	1	0
	3	1	0	0	1	1	1	1	0	1	1
	4	1	1	0	1	1	1	1	1	1	1

Fig. 3.11. Illustration of the word RAM algorithm for the subset sum problem, using the instance $\{w_1, w_2, w_3, w_4\} = \{3, 5, 1, 4\}$, $c = 9$. Table $z_j(\overline{w})$ is the ordinary Bellman recursion, while $g(j, \overline{w})$ is used for the word RAM recursion.

Initially we set $g(0, \overline{w}) := 0$ for $\overline{w} = 0, \ldots, c$ and then we may use the recursion

$$g(j, \overline{w}) = 1 \Leftrightarrow \Big(g(j-1, \overline{w}) = 1 \quad \vee \quad g(j-1, \overline{w} - w_j) = 1 \Big) \tag{3.13}$$

for $j = 1, \ldots, n$ and $\overline{w} = 0, \ldots, c$.

As we only need one bit for representing each entry in $g(j, \overline{w})$ we may use word encoding to store W entries in each integer word. This means that table $g(j, \cdot)$ takes up $c/\log W \leq c/\log c$ space. To derive $g(j, \cdot)$ from $g(j-1, \cdot)$ we simply copy $g(j-1, \cdot)$ to a new table $g'(j-1, \cdot)$ which is shifted right by w_j bits. Then we merge the two lists $g(j-1, \cdot)$ and $g'(j-1, \cdot)$ through binary `or` obtaining $g(j, \cdot)$. The whole operation can be done in $O(c/\log c)$ time as each word can be shifted in constant time. Since we must perform n iterations of recursion (3.13) we get the time complexity $O(nc/\log c)$.

Notice that since the algorithm is based on (2.8), we solve the *all-capacities* version of the problem as introduced in Section 1.3. Hence the table $g(n, \overline{w})$ will hold all optimal solution values for capacities $\overline{w} = 0, \ldots, c$. It should also be mentioned that the algorithm does not make use of any multiplications. Hence, we work on the

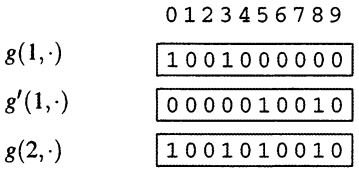


Fig. 3.12. The figure shows how to get from $g(1, \cdot)$ to $g(2, \cdot)$ through a right shift with $w_2 = 5$ positions and a binary `or`. The considered instance is the same as in Figure 3.11.

restricted RAM model where only AC^0 instructions are used. Moreover, no so-called *native constants* are needed (see Hagerup [205] for details).

In Section 4.1.1 the resulting word-RAM algorithm will be presented in detail and the time and space complexity will be proven. Extensive computational results will be presented,showing that the speed-up obtained by using the word RAM algorithm in comparison to recursion (2.8) is significantly larger than W. In the subset sum problem we were in the lucky situation that simple machine instructions (a binary shift right followed by a binary or) can be used to get from list $g(j-1,\cdot)$ to $g(j,\cdot)$. When dealing with the knapsack problem in Section 5.2.1 we are not in this lucky situation. We will instead somehow "build our own instruction set" for the given purpose. The instruction set is implemented by a table look-up, where the size of the table does not become larger than the space used for the ordinary recursion.

3.8 Relaxations

As described in Section 3.7, branch-and-bound algorithms rely on the ability to derive tight upper bounds which may be used to prune the search tree. The most obvious technique for deriving an upper bound is the LP-relaxation as introduced in Section 2.2 where the integrality constraint on the decision variables is simply removed. Other bounding techniques include *Lagrangian relaxation* and *surrogate relaxation*. We will frequently denote the LP-relaxation of a problem P by $C(P)$, the Lagrangian relaxation by $L(P,\lambda)$ and the surrogate relaxation by $S(P,\mu)$. If it is clear from the context, we will sometimes omit the P, i.e. we will write $L(\lambda)$ instead of $L(P,\lambda)$ and so on.

To briefly introduce the two latter techniques let us consider an integer programming problem in general form.

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{3.14}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i = 1,\ldots,m, \tag{3.15}$$

$$\sum_{j=1}^{n} w'_{ij} x_j \leq c'_i, \quad i = 1,\ldots,m', \tag{3.16}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

If we assume somehow that the problem is easier to solve without the first m constraints (3.15), it would be natural to relax these constraints. Thus let $\lambda = (\lambda_1,\ldots,\lambda_m)$ be a vector of nonnegative *Lagrangian multipliers*.

The *Lagrangian relaxed problem* $L(P,\lambda)$ becomes

$$z(L(P,\lambda)) = \text{maximize} \sum_{j=1}^{n} p_j x_j - \sum_{i=1}^{m} \lambda_i \left(\sum_{j=1}^{n} w_{ij} x_j - c_i \right) \tag{3.17}$$

$$\text{subject to} \sum_{j=1}^{n} w'_{ij} x_j \leq c'_i, \quad i = 1,\ldots,m', \tag{3.18}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

The relaxed problem $L(P,\lambda)$ does not contain the unpleasant constraints (3.15) which are included in the objective function (3.17) as a *penalty term*

$$\sum_{i=1}^{m} \lambda_i \left(\sum_{j=1}^{n} w_{ij} x_j - c_i \right). \tag{3.19}$$

Violations of the constraints (3.15) make the penalty term positive and produce a reduction of the objective function.

All feasible solutions to (3.14) are also feasible solutions to (3.17). The objective value of feasible solutions to (3.14) is not larger than the objective value in (3.17), because $\sum_{j=1}^{n} w_{ij} x_j - c_i \leq 0$ for $i = 1,\ldots,m$. Thus, the optimal solution value to the relaxed problem (3.17) is an upper bound to the original problem (3.14) for any vector of nonnegative multipliers.

In a branch-and-bound algorithm we are interested in achieving the tightest upper bound in (3.17). Hence, we would like to choose a vector of nonnegative multipliers λ such that (3.17) is minimized. This leads to the *Lagrangian dual problem*

$$z(LD(P)) = \min_{\lambda \geq 0} z(L(P,\lambda)). \tag{3.20}$$

The Lagrangian dual problem $LD(P)$ yields the least upper bound available from all Lagrangian relaxations. The problem of finding an optimal vector of multipliers λ in $LD(P)$ can be stated as a linear programming problem (see e.g. Nemhauser and Wolsey [360, Section II.3]). Let Y denote the set of feasible solutions of $L(P,\lambda)$. If $C(L(P,\lambda))$ is identical to the convex hull of Y, then the optimal multipliers λ are identical to the optimal values of the dual variables of $C(P)$. In a branch-and-bound algorithm one will often be satisfied with a sub-optimal choice of multipliers λ if only the bound can be derived quickly. In this case subgradient optimization techniques as described by Nemhauser and Wolsey [360, Section I.2] can be applied. More advanced techniques include *surrogate subgradient methods* (see e.g. Kaskavelis and Caramanis [261]), *bundle methods* (see e.g. Kiwiel [271]) or *trust region methods* (see e.g. Marquardt [320]).

To see how Lagrangian relaxation can be used to derive bounds for knapsack type problems we consider the multiple knapsack problem (MKP) as introduced in Section 1.2. If we relax the constraints (1.9) using multipliers $\lambda_1,\ldots,\lambda_n \geq 0$ we get

$$\text{maximize} \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} - \sum_{j=1}^{n}\lambda_j \left(\sum_{i=1}^{m} x_{ij} - 1\right) \tag{3.21}$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i = 1,\ldots,m, \tag{3.22}$$

$$x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m,\ j = 1,\ldots,n.$$

By setting $\tilde{p}_j := p_j - \lambda_j$ for $j = 1,\ldots,n$ the relaxed problem can be decomposed into m independent knapsack problems (KP), where problem i has the form

$$\begin{aligned} z_i = \text{maximize} & \sum_{j=1}^{n} \tilde{p}_j x_{ij} \\ \text{subject to} & \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \\ & x_{ij} \in \{0,1\}, \quad j = 1,\ldots,n, \end{aligned}$$

for $i = 1,\ldots,m$. All the problems have similar profits and weights, thus only the capacities distinguish the individual instances. An optimal solution to (3.21) is finally found as $\sum_{i=1}^{m} z_i + \sum_{j=1}^{n} \lambda_j$.

A different relaxation technique is the *surrogate relaxation*. Returning back to the general problem (3.14) we will again assume that the first m constraints (3.15) are somehow difficult to handle. Instead of removing them from the set of constraints, we can also simplify the problem by merging them into a single constraint. This is done by a linear combination. Choosing a vector of nonnegative multipliers $\mu = (\mu_1,\ldots,\mu_m)$ we get the *surrogate relaxed problem* $S(P,\mu)$

$$z(S(P,\mu)) = \text{maximize} \sum_{j=1}^{n} p_j x_j \tag{3.23}$$

$$\text{subject to} \sum_{i=1}^{m}\mu_i \sum_{j=1}^{n} w_{ij} x_j \leq \sum_{i=1}^{m} \mu_i c_i, \tag{3.24}$$

$$\sum_{j=1}^{n} w'_{ij} x_j \leq c'_i, \quad i = 1,\ldots,m',$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

As all feasible solutions to (3.14) are also feasible solutions to the relaxed problem (3.23) we conclude that $z(S(P,\mu))$ is an upper bound to the original problem (3.14) for all nonnegative multipliers μ.

A possible surrogate relaxation of (MKP) is achieved by relaxing e.g. the constraints (1.8) using multipliers $(\mu_1,\ldots,\mu_m)$.

$$\begin{aligned}
&\text{maximize} \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \\
&\text{subject to} \sum_{i=1}^{m}\mu_i \sum_{j=1}^{n} w_j x_{ij} \le \sum_{i=1}^{m}\mu_i c_i, \\
&\qquad \sum_{i=1}^{m} x_{ij} \le 1, \quad j=1,\ldots,n, \\
&\qquad x_{ij} \in \{0,1\}, \quad i=1,\ldots,m,\ j=1,\ldots,n.
\end{aligned} \tag{3.25}$$

The best choice of multipliers μ for (MKP) are those producing the smallest objective value in (3.25). This leads to the *surrogate dual problem*

$$z(SD(P)) = \min_{\lambda \ge 0} z(S(P,\lambda)). \tag{3.26}$$

Martello and Toth [328] proved that for (MKP) the optimal choice of multipliers is to set them all to the same value, i.e. $\mu_i = k$, $i = 1,\ldots,m$, for a positive constant k. Choosing μ in this way we obtain the standard knapsack problem (KP)

$$\begin{aligned}
&\text{maximize} \sum_{j=1}^{n} p_j x'_j \\
&\text{subject to} \sum_{j=1}^{n} w_j x'_j \le c, \\
&\qquad x'_j \in \{0,1\}, \quad j=1,\ldots,n,
\end{aligned}$$

where the introduced decision variables $x'_j := \sum_{i=1}^{m} x_{ij}$ indicate whether item j is chosen for any of the knapsacks $i = 1,\ldots,m$, and $c := \sum_{i=1}^{m} c_i$ is the sum of all the capacities. We refer to Section 10.2 for more details.

In Nemhauser and Wolsey [360, Section II.3] it is shown that the Lagrangian relaxed problem with the optimal choice of multipliers λ, leads to a bound which is not larger than the bound obtained by LP-relaxation. Moreover the surrogate relaxed problem with the optimal choice of multipliers μ leads to a bound which is not larger than the one obtained by Lagrangian relaxation.

3.9 Lagrangian Decomposition

A final relaxation technique is *Lagrangian decomposition*. The idea in Lagrangian decomposition is to split the problem into a number of independent problems which can be solved (more) efficiently. Typically, a variable x_j which occurs in several

places in the model is replaced with a number of copy variables x_j^i that are linked to the original variable through constraints $x_j^i = x_j$. By relaxing the latter constraint in a Lagrangian way, one hopes to get a number of independent problems formulated in the x_j^i variables. The multiplier associated with the relaxed constraint can then be used to bind together the individual problems.

To be more specific, consider as in Section 3.8 an integer programming model of the form

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i = 1,\ldots,m, \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned} \tag{3.27}$$

Introducing copy variables x_j^i for each variable x_j and linking them to e.g. x_j^1 we get the model

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j^1 \\ \text{subject to} \quad & \sum_{j=1}^{n} w_{ij} x_j^i \leq c_i, \quad i = 1,\ldots,m, \\ & x_j^1 = x_j^i, \quad i = 2,\ldots,m,\ j = 1,\ldots,n, \\ & x_j^i \in \{0,1\}, \quad i = 1,\ldots,m,\ j = 1,\ldots,n. \end{aligned} \tag{3.28}$$

Now relaxing the sum of the constraints from (3.28) in a Lagrangian way using multipliers $\lambda_i \in \mathbb{R}$ we get

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j^1 - \sum_{i=2}^{m} \lambda_i \sum_{j=1}^{n} (x_j^1 - x_j^i) \qquad (3.29) \\ = \quad & \sum_{j=1}^{n} \left(p_j - \sum_{i=2}^{m} \lambda_i \right) x_j^1 + \sum_{i=2}^{m} \lambda_i \sum_{j=1}^{n} x_j^i \\ \text{subject to} \quad & \sum_{j=1}^{n} w_{ij} x_j^i \leq c_i, \quad i = 1,\ldots,m, \qquad (3.30) \\ & x_j^i \in \{0,1\}, \quad i = 1,\ldots,m,\ j = 1,\ldots,n. \end{aligned}$$

The problem can now be decomposed into m independent knapsack problems of the form

$$\text{maximize } z_i = \sum_{j=1}^{n} \tilde{p}_j^i x_j^i$$

$$\text{subject to } \sum_{i=1}^{n} w_{ij} x_j^i \leq c_i\,,$$

$$x_j^i \in \{0,1\}, \quad j = 1,\ldots,n$$

where

$$p_j^i := \begin{cases} p_j - \sum_{k=2}^{m} \lambda_k & \text{for } i = 1 \\ \lambda_i & \text{for } i = 2,\ldots,m. \end{cases} \tag{3.31}$$

The overall solution is found as $z := \sum_{i=1}^{m} z_i$.

3.10 The Knapsack Polytope

The extensive study of the knapsack polytope is not a topic of this book. For the sake of completeness a short survey on the principal concepts and basic results will be presented in this chapter. The books by Nemhauser and Wolsey [360] and Schrijver [428] contain a detailed introduction to the theory of polyhedra. For a comprehensive review of the general theory of polyhedral properties of the knapsack problem we refer to Weismantel [480]. Our survey starts with fundamental prerequisites from linear algebra:

Let $x^1,\ldots,x^k \in \mathbb{R}^n$ be n-dimensional vectors. A vector x is called a *linear combination* of $x^1,\ldots,x^k$ if there are $\lambda_1,\ldots,\lambda_k \in \mathbb{R}$ such that $x = \sum_{j=1}^{k} \lambda_j x^j$ holds. The vectors $x^1,\ldots,x^k$ are called *linearly independent* if $\sum_{j=1}^{k} \lambda_j x^j = 0$ always implies $\lambda_1 = \ldots = \lambda_k = 0$, otherwise they are called *linearly dependent*. (We will identify 0 with the zero vector $(0,\ldots,0)$ if it is clear from the context.) $x^1,\ldots,x^k$ are linearly dependent if and only if there is at least one $x^i \in \{x^1,\ldots,x^k\}$ which can be expressed as linear combination of the other vectors.

A vector x is called an *affine combination* of $x^1,\ldots,x^k$ if there are $\lambda_1,\ldots,\lambda_k \in \mathbb{R}$ such that $x = \sum_{j=1}^{k} \lambda_j x^j$ and $\sum_{j=1}^{k} \lambda_j = 1$ hold. $x^1,\ldots,x^k$ are called *affinely independent* if $\sum_{j=1}^{k} \lambda_j x^j = 0$ and $\sum_{j=1}^{k} \lambda_j = 0$ always implies $\lambda_1 = \ldots = \lambda_k = 0$, otherwise they are called *affinely dependent*. This means, $x^1,\ldots,x^k$ are affinely independent if and only if none of the points is an affine combination of the others. Alternatively, $x^1,\ldots,x^k$ are affinely independent if and only if $x^2 - x^1,\ldots,x^k - x^1$ are linearly independent. The maximum number of affinely independent vectors in $\mathbb{R}^n$ is $n+1$ whereas the maximum number of linearly independent vectors in $\mathbb{R}^n$ is n.

A vector x is called a *convex combination* of $x^1,\ldots,x^k$ if there are nonnegative $\lambda_1,\ldots,\lambda_k \in \mathbb{R}$ such that $x = \sum_{j=1}^{k} \lambda_j x^j$ and $\sum_{j=1}^{k} \lambda_j = 1$ hold. A set $S \subseteq \mathbb{R}^n$ is called *convex* if for all $x^1, x^2 \in S$ and any $0 \leq \lambda \leq 1$ also the convex combination

$\lambda x^1 + (1-\lambda)x^2$ belongs to S. The *convex hull* of a set $S \subseteq \mathbb{R}^n$ is the set of all convex combinations of elements of S and will be denoted by $\text{conv}(S)$.

A *polyhedron* $P \subseteq \mathbb{R}^n$ is a set of vectors which satisfy finitely many linear inequalities. Thus, P can be described as the set of solutions of the linear inequality system $Ax \leq b$ where A is an $m \times n$ matrix and $b \in \mathbb{R}^m$, i.e. $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$. A bounded polyhedron is called *polytope*. A polyhedron P has *dimension* k if the maximum number of affinely independent elements of P is $k+1$, shortly $\dim(P) = k$. A *full-dimensional polyhedron* has dimension n.

For a given polyhedron P it is interesting to know which of the satisfied inequalities are really necessary for describing P. This motivates the definition of the facet of a polyhedron. An inequality $a_1x^1 + \ldots + a_nx^n \leq b$ (shortly: $ax \leq b$) is a *valid inequality* for the polyhedron P if each $x \in P$ satisfies the inequality. The corresponding set $F = \{x \in P \mid ax = b\}$ is called a *face* of P and the inequality $ax \leq b$ *defines* F, i.e. $ax \leq b$ is a *face defining inequality*. The face F is *proper* if $\emptyset \neq F \neq P$. If F is non-empty then F is a *supporting face* of F. A direct consequence is that for any proper face F, $\dim(F) < \dim(P)$. A face with maximal dimension, $\dim(F) = \dim(P) - 1$, is said to be a *facet*. Thus, a facet contains exactly $\dim(P) - 1$ affinely independent elements of the corresponding polyhedron P. It can be shown (see e.g. Nemhauser and Wolsey [360, Section 1.4]) that the set of facets are necessary and sufficient for the description of a polyhedron P, i.e. the facets represent the minimal inequality representation of P. Especially, if P is full-dimensional, then there is a unique minimal set of facets defining P.

The interest in studying facets is based on the following fundamental fact: Let $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ be a polyhedron and S be the set of integer valued vectors in S. Then the solution of the integer program $\max\{px \mid x \in S\}$ is equivalent to solving the linear program $\max\{px \mid x \in \text{conv}(S)\}$. Therefore, the facet defining inequalities are exactly those inequalities which are required to define the convex hull of the feasible solutions.

Now we turn to the knapsack problem: The *knapsack polytope* P is the convex hull of the set of all solution vectors for the knapsack problem, i.e.

$$P := \text{conv}\left\{ x \in \mathbb{R}^n \;\middle|\; \sum_{j=1}^{n} w_j x_j \leq c, x_j \in \{0,1\}, j = 1,\ldots,n \right\}. \tag{3.32}$$

Define e^j to be the j-th unit vector, i.e. $e^j_j = 1$ and $e^j_i = 0$ for $i \neq j$. Due to the assumption $w_j \leq c$ (1.14) on the item weights, the elements of the set $V := \{e^1, \ldots e^n, 0\}$ are affinely independent solution vectors of the knapsack problem. Therefore, P has full dimension $\dim(P) = n$. Simple facet defining inequalities of P are the inequalities $x_j \geq 0$ for $j = 1,\ldots,n$. This can be seen by removing e^j from V, i.e. considering $V \setminus \{e^j\}$. All remaining vectors are elements of $F = \{x \in P \mid x_j = 0\}$ and affinely independent.

The study of the facets of the knapsack polytope dates back to Balas [18], Hammer, Johnson and Peled [207] and Wolsey [488] who investigated an important class of valid inequalities for the knapsack problem, the so-called minimal cover inequalities:

Let $S \subseteq N$ be a subset of the ground set N. We call S a *cover (set)* or a *dependent set* for P if $\sum_{j \in S} w_j > c$ holds. S is a *minimal cover (set)* for P if

$$\sum_{j \in S \setminus \{i\}} w_j \leq c \tag{3.33}$$

holds for all $i \in S$. This means that the removal of any single item j for S will eliminate the cover property. It can be easily seen that for a minimal cover S the *minimal cover inequality*

$$\sum_{j \in S} x_j \leq |S| - 1 \tag{3.34}$$

provides a valid inequality for P (see also Section 15.3). Let $P_{\tilde{N}}$ denote the knapsack polytope defined for the item set $\tilde{N} \subseteq N$. For $s \in S$ let $x^s \in \mathbb{R}^{|\tilde{N}|}$ be defined as

$$x_j^s := \begin{cases} 0, & j = s, \\ 1, & \text{otherwise}. \end{cases}$$

Clearly, the vectors x^s, $s \in S$, are affinely independent. Thus, the minimal cover inequality defines a facet of the knapsack polytope P_S.

In general, the minimal cover inequalities (3.34) are not facet-defining for P, but they can be *lifted* to define a facet of the knapsack polytope. One example for a lifted cover inequality is obtained by introducing the *extension* $E(S)$ for a minimal cover S which is defined as

$$E(S) := S \cup \{j \in N \setminus S \mid w_j \geq w_i,\ i \in S\}. \tag{3.35}$$

Then the generalization of (3.34) to the *extended cover inequality*

$$\sum_{j \in E(S)} x_j \leq |S| - 1 \tag{3.36}$$

provides also a valid inequality. (An example can be found at the end of this section.)

Balas [18], Hammer, Johnson and Peled [207] and Wolsey [488] gave necessary and sufficient conditions for (3.36) to define a facet of the knapsack polytope. A quite general form of a facet arising from minimal covers is obtained by partitioning a minimal cover S for P into two subsets S_1, S_2 with $S = S_1 \cup S_2$ and $S_1 \neq \emptyset$. Then $\text{conv}(P)$ has a facet defined by an inequality of the form

$$\sum_{j \in N \setminus S} a_j + \sum_{j \in S_2} b_j x_j + \sum_{j \in S_1} x_j \leq |S_1| - 1 + \sum_{j \in S_2} b_j \tag{3.37}$$

with $a_j \geq 0$ for all $j \in N \setminus S$ and $b_j \geq 0$ for all $j \in S_2$.

A minimal cover S is called a *strong minimal cover* if either $E(S) = N$ or

$$\sum_{j \in S \setminus \{\alpha\}} w_j + w_\beta \leq c \tag{3.38}$$

with $\alpha := \text{argmax}\{w_j \mid j \in S\}$ and $\beta := \text{argmax}\{w_j \mid j \in N \setminus E(S)\}$. An $O(n \log n)$ procedure for computing facets of the knapsack polytope by lifting the inequalities induced by the extensions of *strong* minimal covers has been recently presented by Escudero, Garín and Pérez [133].

A generalization of (3.34) is given by the so-called $(1,k)$-*configuration inequalities* investigated by Padberg [366]. A set $N_1 \cup \{\alpha\}$ with $N_1 \subseteq N$ and $\alpha \in N \setminus N_1$ is called a $(1,k)$-*configuration* if $\sum_{j \in N_1} w_j \leq c$ and $K \cup \{\alpha\}$ is a minimal cover, for all $K \subseteq N_1$ with $|K| = k$. For a given $(1,k)$-configuration $N_1 \cup \{\alpha\}$ with $R \subseteq N_1$ and $r := |R| \geq k$ the inequality

$$(r-k+1)x_\alpha + \sum_{j \in R} x_j \leq r, \tag{3.39}$$

is called the $(1,k)$-*configuration inequality* corresponding to $N_1 \cup \{\alpha\}$ and $R \subseteq N_1$. The $(1,k)$-configuration inequality is a valid inequality for $\text{conv}(P)$. For $k = |N_1|$ the $(1,k)$-configuration inequality is reduced to the minimal cover inequality (3.34). Padberg [366] showed that the complete set of facets of the polytope $P_{N_1 \cup \{\alpha\}}$ is determined by the $(1,k)$-configuration inequalities.

Another general idea for obtaining facets is the concept of *lifting* which we have already mentioned in the context of an extension for a minimal cover. The concept of lifting is due to Padberg [365]. We have already seen that minimal cover inequalities and $(1,k)$-configuration inequalities define facets of lower dimensional polytopes. The idea is to extend them into $\mathbb{R}^n$ so as to yield facets or valid inequalities for the knapsack polytope P. Formally, let $S \subset N$ and $ax \leq b$ be a valid inequality for the polytope P_S. The inequality $a'x \leq b$ is called a *lifting* of $ax \leq b$ if $a'_j = a_j$ holds for all $j \in S$.

When applying lifting to the minimal cover inequalities (3.34) one gets inequalities of the form

$$\sum_{j \in S} x_j + \sum_{j \in N \setminus S} \alpha_j x_j \leq |S| - 1. \tag{3.40}$$

The procedure of Padberg is sequential, in that the lifting coefficients α_j are computed one by one in a given sequence. Thus, it is also called a *sequential lifting procedure*. A disadvantage of the approach by Padberg is that a certain integer program must be solved to optimality for the computation of each coefficient α_j. It was shown by Zemel [499] that these computations can be done in $O(n|S|)$ time.

Balas and Zemel [21] provided a *simultaneous lifting procedure* which calculates the facets obtained from minimal covers. The results of Balas and Zemel have been

generalized by Nemhauser and Vance [359]. In [23] Balas and Zemel show that using lifting and "complementing", all facet defining inequalities of the positive 0-1 polytopes can be obtained from minimal covers. A characterization of the entire class of facets for knapsack polytopes with small capacity values are given by Hammer and Peled [209]. Results concerning the computational complexity of obtaining lifted cover inequalities can be found in Hartvigsen and Zemel [214].

Another class of inequalities, the so-called *weight inequalities*, were proposed by Weismantel [481]. Assume that the subset $T \subseteq N$ satisfies $\sum_{j\in T} w_j < c$ and define the residual capacity as $r = c - \sum_{j\in T} w_j$. The weight inequality with respect to T is defined by

$$\sum_{j\in T} w_j x_j + \sum_{j\in N\setminus T} \max\{0, w_j - r\} x_j \leq \sum_{j\in T} w_j. \tag{3.41}$$

Weismantel [481] proved that the inequality is valid for P.

Extended weight inequalities are defined as follows. Let $T_1 \subseteq N$ and $T_2 \subseteq N$ be two disjoint subsets satisfying $\sum_{j\in T_1\cup T_2} w_j \leq c$, and $w_i \leq w_j$ for all $i \in T_1$ and $j \in T_2$ for which the inequalities $\sum_{i\in T_1} w_i \geq w_j$ for all $j \in T_2$ hold. Define the relative weight of an item $k \in T_1 \cup T_2$ as

$$\overline{w}_k := \begin{cases} 1 & \text{if } k \in T_1, \\ \min\{|S| \mid S \subseteq T_1, \sum_{j\in S} w_j \geq w_k\} & \text{if } k \in T_2. \end{cases} \tag{3.42}$$

Under these assumptions we define for an item $h \in N \setminus (T_1 \cup T_2)$ the extended weight inequality

$$\sum_{j\in T_1} x_j + \sum_{j\in T_2} \overline{w}_j x_j + \overline{w}_h x_h \leq |T_1| + \sum_{j\in T_2} \overline{w}_j, \tag{3.43}$$

where $\overline{w}_h = \min\{|S_1| + \sum_{j\in S_2} \overline{w}_j \mid S_1 \subseteq T_1, S_2 \subseteq T_2, \sum_{j\in S_1\cup S_2} w_j \geq w_h - r\}$ and $r = c - \sum_{j\in T_1\cup T_2} w_j$. Weismantel [481] proved that (3.43) is valid for P and that lifting coefficients can always be computed in polynomial time.

Example: (cont.) Recall the example of Section 2.1. Then the knapsack polytope is given as

$$P = \text{conv}\{x \in \{0,1\}^7 \mid 2x_1 + 3x_2 + 6x_3 + 7x_4 + 5x_5 + 9x_6 + 4x_7 \leq 9\}.$$

$S_1 = \{1,2,3\}$ and $S_2 = \{4,5\}$ are examples for minimal covers of P. The corresponding minimal cover inequalities are

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 2, \\ x_4 + x_5 &\leq 1. \end{aligned}$$

The extended cover inequalities for S_1 and S_2 are

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 &\leq 2, \\ x_3 + x_4 + x_5 + x_6 &\leq 1. \end{aligned}$$

The minimal cover $x_4 + x_5 \leq 1$ is not facet defining. But the extension $x_3 + x_4 + x_5 + x_6 \leq 1$ defines the facet $F := \{x \in P \mid x_3 + x_4 + x_5 + x_6 = 1\}$ since e^3, e^4, e^5, e^6, $e^1 + e^5$, $e^2 + e^5$, $e^7 + e^5$ are affinely independent elements of F.

Set $N_1 = \{1,2\}$ and $\alpha = 6$. Then, $\{1,2\} \cup \{6\}$ is a $(1,1)$-configuration. The corresponding $(1,1)$-configuration inequalities

$$x_1 + x_6 \leq 1, \qquad x_2 + x_6 \leq 1, \qquad x_1 + x_2 + 2x_6 \leq 2,$$

define facets of the polytope $P_{\{1,2,6\}}$. □

4. The Subset Sum Problem

Given a set $N = \{1,\dots,n\}$ of n items with positive integer weights $w_1,\dots,w_n$ and a capacity c, the *subset sum problem* (SSP) is to find a subset of N such that the corresponding total weight is maximized without exceeding the capacity c. Recall the formal definition as introduced in Section 1.2:

$$\text{(SSP)} \qquad \text{maximize} \quad \sum_{j=1}^{n} w_j x_j \tag{4.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c, \tag{4.2}$$

$$x_j \in \{0,1\}, \quad j = 1,\dots,n. \tag{4.3}$$

We will assume that assumptions (1.14) to (1.16) as defined for (KP) also hold for (SSP). This means that every item j fits into the knapsack, i.e.,

$$w_j \le c, \quad j = 1,\dots,n, \tag{4.4}$$

and that the overall weight sum of the items exceeds c, i.e.,

$$\sum_{j=1}^{n} w_j > c. \tag{4.5}$$

Without loss of generality we assume that all weights are positive, i.e.,

$$w_j > 0, \quad j = 1,\dots,n, \tag{4.6}$$

since otherwise we may use the transformation from Section 1.4 to achieve this assumption.

We will often identify the items with their corresponding weights. (SSP) can be considered as a special case of the knapsack problem arising when the profit and the weight associated with each item are identical. (SSP) has numerous applications: Solutions of subset problems can be used for designing better lower bounds for scheduling problems (see Guéret and Prins [200] and Hoogeveen et al. [237]).

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

The constraints of 0-1 integer programs could be tightened by solving subset sum problems with some additional constraints (see e.g. Dietrich and Escudero [105] and Escudero, Martello and Toth [134]), and it appears as subproblem in numerous combinatorial problems (see e.g. Pisinger [386]).

Several authors considered the idea of applying the subset sum problem as a subprocedure in a bin packing algorithm. This means that items are packed into bins filling one bin at a time, each as much as possible. Recently, Caprara and Pferschy [68] managed to show that the resulting heuristic has a worst-case performance ratio between 1.6067... and 1.6210.... Variants of this approach were considered in Caprara and Pferschy [67].

(SSP) formulated as a *decision problem* asks whether there exists a subset of N such that the corresponding weights add up exactly to the capacity c. To distinguish it from the optimization problem we denote it as SSP-DECISION (see also Appendix A). The decision problem is of particular interest in cryptography since SSP-DECISION with unique solutions corresponds to a secret message to be transmitted. We will go into details with cryptosystems based on subset sum problems in Section 15.6.

Although (SSP) is a special case of (KP) it is still $\mathcal{NP}$-hard [164] as will be shown in Appendix A. Clearly, (SSP) can be solved to optimality in pseudopolynomial time by the dynamic programming algorithms described in Section 2.3. But due to the simple structure of (SSP) adapted algorithms can have much better behavior. For (SSP) all upper bounds based on some kind of continuous relaxation as they will be described in Section 5.1.1, give the trivial bound $U = c$. Thus, although (SSP) in principle can be solved by every branch-and-bound algorithm for (KP), the lack of tight bounds may imply an unacceptably large computational effort. Therefore, it is worth constructing algorithms designed specially for (SSP). Also, owing to the lack of tight bounds, it may be necessary to apply heuristic techniques to obtain a reasonable solution within a limited time.

In this chapter exact and approximation algorithms for the (SSP) are investigated. We will start the treatment of (SSP) in Section 4.1 dealing with different dynamic programming algorithms: If the coefficients are not too large, dynamic programming algorithms are generally able to solve subset sum problems even when the decision problem SSP-DECISION has no solution. In Section 4.2.1 upper bounds tighter than the trivial bound $U = c$ are considered. In Section 4.2 we will present some hybrid algorithms which combine branch-and-bound with dynamic programming. Section 4.3 shows how large-sized instances can be solved by defining a core problem, and Section 4.4 gives a computational comparison of the exact algorithms presented. Section 4.5 deals with polynomial time approximation schemes for (SSP). In Section 4.6 we will present a new *FPTAS* for (SSP) which is superior to all other approximation schemes and runs in $O(\min\{n \cdot 1/\varepsilon, n + 1/\varepsilon^2 \log(1/\varepsilon)\})$ time and requires only $O(n + 1/\varepsilon)$ space. We will close this chapter with a study on the computational behavior of the new *FPTAS*.

4.1 Dynamic Programming

Because of the lack of tight bounds for (SSP) dynamic programming may be the only way of obtaining an optimal solution in reasonable time when there is no solution to the decision problem SSP-DECISION. In addition, dynamic programming is frequently used to speed up branch-and-bound algorithms by avoiding a repeated search for sub-solutions with the same capacity.

The Bellman recursion (2.8) presented in Section 2.3 for (KP) is trivially simplified to the (SSP): At any stage we let $z_j(d)$, for $0 \leq j \leq n$ and $0 \leq d \leq c$, be an optimal solution value to the subproblem of (SSP) defined on items $1,\ldots,j$ with capacity d. Then the Bellman recursion becomes

$$z_j(d) = \begin{cases} z_{j-1}(d) & \text{for } d = 0,\ldots,w_j - 1, \\ \max\{z_{j-1}(d),\ z_{j-1}(d-w_j)+w_j\} & \text{for } d = w_j,\ldots,c, \end{cases} \tag{4.7}$$

where $z_0(d) = 0$ for $d = 0,\ldots,c$. The resulting algorithm is called Bellman.

Obviously, we may use any of the dynamic programming algorithms presented in Section 2.3 to solve (SSP) to optimality in pseudopolynomial time. Of particular interest is algorithm DP-with-Lists introduced in Section 3.4 because it can be easily transformed into the well-known algorithm by Bellman for (SSP) [32]. Bellman's approach works in the following way: The set R of *reachable values* consists of integers d less than or equal to the capacity c for which a subset of items exists with total weight equal to d. Starting from the empty set, R is constructed iteratively in n iterations by adding in iteration j weight w_j to all elements from R and keeping only partial sums not exceeding the capacity. For each value $d \in R$ a corresponding *solution set* $X(d)$ with total weight equal to d is stored. Finally, z^* is the the maximum value of R and the optimum solution set $X^* = X(z^*)$. This gives a pseudopolynomial algorithm with time $O(nc)$ and space $O(nc)$. algorithm Bellman-with-Lists is depicted in Figure 4.1.

Algorithm Bellman-with-Lists:

$R_0 := \{0\}$
for $d := 0$ **to** c **do**
  $X(d) := \emptyset$ *initialization*
for $j := 1$ **to** n **do**
  $R'_{j-1} := R_{j-1} \oplus w_j$ *add w_j to all elements in R_{j-1}*
  delete all elements $\overline{w} \in R'_{j-1}$ **with** $\overline{w} > c$
  $R_j := R_{j-1} \cup R'_{j-1}$
  for all $i \in R_j \setminus R_{j-1}$ **do**
    $X(i) := X(i-w_j) \cup \{j\}$
$R := R_n$, $z^* := \max\{d \mid d \in R\}$, $X^* := X(z^*)$

Fig. 4.1. Algorithm Bellman-with-Lists for (SSP) based on DP-with-Lists.

Algorithm Bellman-with-Lists is of course inferior to algorithm Recursive-DP of Section 3.3 when applied to the knapsack problem. By Corollary 3.3.2 Recursive-DP produces an optimal solution in $O(nc)$ time and $O(n+c)$ space, but has a rather complicated structure. For (SSP) an algorithm with the same time and space requirements can be given in a much easier way. Instead of storing the whole set $X(d)$ we record only the most recently added item $r(d)$ for every element of R. In contrary to algorithm DP-3 of Section 2.3 which restarts the algorithm again with reduced capacity, we can determine X^* by collecting the items in a simple backtracking routine. The resulting algorithm Improved-Bellman is described in Figure 4.2.

Algorithm Improved-Bellman:

$R_0 := \{0\}$
for $d := 0$ to c do
    $r(d) := 0$ *initialization*
for $j := 1$ to n
    $R'_{j-1} := R_{j-1} \oplus w_j$ *add w_j to all elements in R_{j-1}*
    delete all elements $\overline{w} \in R'_{j-1}$ with $\overline{w} > c$
    $R_j := R_{j-1} \cup R'_{j-1}$
    for all $d \in R_j \setminus R_{j-1}$ do
        $r(d) := j$
$R := R_n$, $z^* := \max\{d \mid d \in R\}$
$\bar{c} := z^*$
repeat
    $X^* := X^* \cup r(\bar{c})$, $\bar{c} := \bar{c} - r(\bar{c})$
until $\bar{c} = 0$ *constructing X^* by backtracking*

Fig. 4.2. Improved-Bellman uses backtracking to improve Bellman-with-Lists.

Theorem 4.1.1. *Algorithm* Improved-Bellman *solves (SSP) to optimality in $O(nc)$ time and $O(n+c)$ space.*

Proof. It is only necessary to show the correctness of the algorithm, which means that no item is collected twice when performing backtracking for the construction of X^*. But this can never happen, because the recently added item $r(d)$ is fixed when we reach the value d for the first time and not changed afterwards. Hence, the indices of the items put into X^* are strictly decreasing while performing backtracking. □

4.1.1 Word RAM Algorithm

As mentioned in Section 3.7 the Bellman recursion (4.7) for the subset sum problem is suitable for word encoding. In this section we will present the resulting word-RAM algorithm in details and prove the time and space complexity. As the algorithm

will make use of some binary operations, we will denote by `shl`, `shr`, `and`, `or` the operations: shift left, shift right, binary and, and binary or (see Corment et al. [92, Section 2.2] for an introduction to to the RAM model of computation). All these instructions can be implemented as constant depth circuits (see e.g. [5]). We will assume that the word size of the computer is W which should be sufficiently large to hold the capacity c, i.e. W is of order $\Theta(\log c)$. As usually, all logarithms are base two.

Define the table $g(j,\overline{w})$ for $j = 1,\ldots,n$ and $\overline{w} = 0,\ldots,c$ as follows

$$\begin{array}{l} g(j,\overline{w}) = 1 \Leftrightarrow \\ \text{there is a subset of } \{w_1,\ldots,w_j\} \text{ with overall sum } \overline{w}. \end{array} \tag{4.8}$$

Obviously $g(0,0) = 1$, and $g(0,\overline{w}) = 0$ for $\overline{w} = 1,\ldots,c$, since the only sum we can obtain with an empty set of items is $\overline{w} = 0$. For the following values of $g(j,\overline{w})$ we may use the recursion

$$g(j,\overline{w}) = 1 \quad \Leftrightarrow \quad \Big(g(j-1,\overline{w}) = 1 \quad \vee \quad g(j-1,\overline{w}-w_j) = 1\Big) \tag{4.9}$$

for $j = 1,\ldots,n$ and $\overline{w} = 0,\ldots,c$. To obtain a more compact representation, we may store W entries of table g as bits in each word, and use binary operations to get from table $g(j-1,\cdot)$ to table $g(j,\cdot)$ as outlined in Figure 4.3. The main loop of the algorithm contains two cases, depending on whether the weight w_j is a multiple of W or not. In the first case, we just need to shift the table g with $v := c/W$ words and `or` it to the existing table (this is done in the last while-loop of the algorithm). If w_j is not a multiple of W, we need to shift the table g with $v := c/W$ words and $a := w_j - v * W$ bits, before it is `or`'ed to the original table (this is done in the second-last while-loop).

Pisinger [392] showed the following time and space bounds for the algorithm.

Proposition 4.1.2 *The table $g(n,\cdot)$ of optimal solution values can be derived in $O(nc/\log c)$ time and $O(n + c/\log c)$ space using recursion (4.9).*

Proof. As we only need one bit for representing each entry in $g(j,\overline{w})$ we may use word encoding to store W entries in each integer word. This means that the table $g(j,\cdot)$ takes up $O(c/\log c)$ space. To derive $g(j,\cdot)$ from $g(j-1,\cdot)$ we simply copy $g(j-1,\cdot)$ to a new table $g'(j-1,\cdot)$ which is shifted right by w_j bits. Then we merge the two lists $g(j-1,\cdot)$ and $g'(j-1,\cdot)$ through binary `or` obtaining $g(j,\cdot)$. The whole operation can be done in $O(c/\log c)$ time since binary `or` of two words can be calculated in constant time, and each word can be shifted in constant time. Notice that if $w_j > W$ then we first shift the table by $\lfloor w_j/W \rfloor$ whole words, followed by a shift of $w_j - \lfloor w_j/W \rfloor$ bits. Assuming that the word size W is a power of two, also the determination of $\lfloor w_j/W \rfloor$ can be implemented by a simple shift operation. Since we must perform n iterations of recursion (4.9) we get the time complexity $O(nc/\log c)$.

```
Algorithm Wordsubsum(n, c, w, g):
S := 0          sum of weights
c̄ := shr(c, B)          index of word holding bit c
for i := 1 to c̄ do g_i := 0          initialize table g
g_0 := 1

for j := 1 to n do          for all items
    v := shr(w_j, B)          number of words to shift
    a := w_j − shl(v, B)          number of bits to shift left
    b := W − a          number of bits to shift right
    S := S + w_j          sum of weights w_1, ..., w_i
    k := min(c̄, shr(S, B))          last position in g to read
    i := k − v          last position in g to write
    if (a ≠ 0) then
        while i > 0 do          main loop if both bit and word shifts
            g_k := or(g_k, shl(g_i, a), shr(g_{i−1}, b))
            i := i − 1, k := k − 1
        g_k := shl(g_i, a)
    else
        while i ≥ 0 do          main loop if only word shifts
            g_k := or(g_k, g_i)
            i := i − 1, k := k − 1
```

Fig. 4.3. Algorithmic description of Wordsubsum. The pseudo-code is very detailed to show that only additions, subtractions, binary `shl`, `shr`, `and`, and `or` are needed. It is assumed that W is the word size and $B = \log W$. The input to the algorithm is a table w of n weights and the capacity c. The output of the algorithm is the table g where a bit will be set at position $\overline{w}$ if the weight sum $\overline{w}$ can be obtained with a subset of the weights $w_1, \ldots, w_n$. Notice that in this implementation bits in a word are numbered from right to left, as opposed to Figure 3.10

The space complexity appears by observing that we only need $g(j-1,\cdot)$ and $g'(j-1,\cdot)$ to derive $g(j,\cdot)$, thus only three arrays are needed. Actually, the whole operation can be done in one single array by considering the values $\overline{w}$ in decreasing order $c,\ldots,0$.

The optimal solution value z^* is found as the largest value $\overline{w} \leq c$ for which $g(n,\overline{w}) = 1$. This value can easily be found in $O(c/\log c)$ time as we start from the right-most word, and repeatedly move leftward skipping whole words as long as they have the value zero. When a word different from zero is identified, we simply use linear search to find the right-most bit. The whole operation takes $O(c/\log c + \log W) = O(c/\log c)$ time. □

Notice that upon termination, table $g(n,\overline{w})$ will hold all solution values for $\overline{w} = 0,\ldots,c$. If we want to find an optimal solution vector x corresponding to any of the solutions $z \leq c$, Pisinger [392] proved the following:

Proposition 4.1.3 *For a given solution value z the corresponding solution vector x which satisfies $\sum_{j=1}^{n} w_j x_j = z$ can be derived in $O(nc/\log c)$ time and $O(n + c/\log c)$ space, i.e., the same complexity as solving the recursion (4.9).*

Proof. We may assume that $z = c$ as we simply can solve the problem once more with capacity equal to z without affecting the time complexity. To find the solution vector x we may apply the divide and conquer framework presented in Section 3.3. The set $N = \{1,\ldots,n\}$ of items is divided into two equally sized parts N_1 and N_2. Without loss of generality we may assume that $|N|$ is even, and thus $|N_1| = |N_2| = n/2$. Each of the problems is now solved separately returning two tables of optimal solution values $g(n/2,\cdot)$ and $g'(n/2,\cdot)$. The first table $g(n/2,\cdot)$ is obtained in the "ordinary" way, while the second table $g'(n/2,\cdot)$ is obtained in a "reverse" way, i.e. we start with $g'(0,c) = 1$ and perform left shifts where we previously performed right shifts. Define table h as a binary and of $g(n/2,\cdot)$ and $g'(n/2,\cdot)$.

If $h(\overline{w}) = 0$ for all $\overline{w} = 0,\ldots,c$ then it is not possible to obtain the solution value c. Otherwise assume that $h(\overline{w}) = 1$ for a specific value of $\overline{w}$ meaning that $g(n/2,\overline{w}) = g'(n/2,c-\overline{w}) = 1$. Hence to obtain the sum c we must find a subset of N_1 which sums to $\overline{w}$ and a subset of N_2 which sums to $c - \overline{w}$. The same approach is used recursively until each subset consists of a single item j, in which case it is trivial to determine whether the corresponding decision variable x_j is 0 or 1. The total time complexity becomes $O(nc/\log c)$ by using equation (3.1). □

4.1.2 Primal-Dual Dynamic Programming Algorithms

Let s be the *split item* for (SSP) with arbitrary ordering of the items, i.e.

$$s = \min\left\{ h : \sum_{j=1}^{h} w_j > c \right\}. \tag{4.10}$$

The *split solution* $\hat{x}$ given by $\hat{x}_j = 1$ for $j = 1,\ldots,s-1$, has weight sum $\hat{w} = \sum_{j=1}^{s-1} w_j$.

The Bellman recursion (4.7) constructs a table of optimal solutions from scratch by gradually extending the problem with new items. Pisinger [387] observed that frequently the optimal solution vector x^* only differs from the split solution $\hat{x}$ for a few items j, and these items are generally located close to the split item s. Hence, a better dynamic programming algorithm should start from the solution $\hat{x}$ and gradually insert or remove some items around s. We will use the name *primal-dual* to denote algorithms which alternate between feasible and infeasible solutions. In linear programming, the term primal-dual is frequently used for algorithms which maintain dual feasibility and complementary slackness conditions, in their search for a primal feasible solution. Let $z_{a,b}(d)$, be an optimal solution to the maximization problem:

$$z_{a,b}(d) = \sum_{j=1}^{a-1} w_j + \max \left\{ \sum_{j=a}^{b} w_j x_j \;\middle|\; \begin{array}{l} \sum_{j=a}^{b} w_j x_j \le d - \sum_{j=1}^{a-1} w_j, \\ x_j \in \{0,1\}, j = a, \ldots, b \end{array} \right\}, \tag{4.11}$$

defined for $a = 1, \ldots, s$, $b = s-1, \ldots, n$ and $0 \le d \le 2c$. In other words $z_{a,b}(d)$ is the optimal solution to a (SSP) with capacity d, defined on items $a, \ldots, b$ and assuming that all items $1, \ldots, a-1$ have been chosen.

The improved algorithm works with two dynamic programming recursions:

$$z_{a,b}(d) = \begin{cases} z_{a,b-1}(d) & \text{if } d - w_b < 0, \\ \max\{z_{a,b-1}(d), z_{a,b-1}(d-w_b) + w_b\} & \text{if } d - w_b \ge 0, \end{cases} \tag{4.12}$$

$$z_{a,b}(d) = \begin{cases} z_{a+1,b}(d) & \text{if } d + w_a > 2c, \\ \max\{z_{a+1,b}(d), z_{a+1,b}(d+w_a) - w_a\} & \text{if } d + w_a \le 2c. \end{cases} \tag{4.13}$$

The first recursion (4.12) relates to the possible insertion of w_b into the knapsack while (4.13) relates to the possible removal of w_a from the knapsack. Initially we set $z_{s,s-1}(d) = -\infty$ for $d = 0, \ldots, \hat{w} - 1$ and $z_{s,s-1}(d) = \hat{w}$ for $d = \hat{w}, \ldots, 2c$. Then we alternate between the two recursions (4.12) and (4.13) evaluating $z_{s,s-1}, z_{s,s}, z_{s-1,s}, z_{s-1,s+1}, \ldots$ until we reach $z_{1,n}$. The optimal solution value is given by $z_{1,n}(c)$. The time complexity of this approach is $O(nc)$ while the space complexity may be reduced to $O(n+c)$ using the framework described in Section 3.3. From a worst-case point of view nothing has been gained compared to the Bellman, but when several solutions to the decision problem SSP-DECISION exists, the recursion may be terminated as soon as $z_{a,b}(c) = c$ for values of a, b close to s, having used time $O((b-a)c)$.

4.1.3 Primal-Dual Word-RAM Algorithm

In the previous section we saw how the Bellman recursion (4.7) can be modified to a primal-dual recursion (4.12) - (4.13). In a similar way, we may modify the word-RAM algorithm of Section 4.1.1 to a primal-dual version.

Let s be the split item defined by (4.10). Define the table $g(a,b,\overline{w})$ for $a \le s \le b$ and $\overline{w} = 0, \ldots, c$ as follows

$$\begin{array}{l} g(a,b,\overline{w}) = 1 \Leftrightarrow \\ \text{there is a subset of } \{w_a, \ldots, w_b\} \text{ with overall sum } \overline{w}. \end{array} \tag{4.14}$$

We set $g(s,s-1,0) = 1$, and $g(s,s-1,\overline{w}) = 0$ for other values of $\overline{w}$. For the following values of $g(a,b,\overline{w})$ we may use the two recursions

$$g(a,b,\overline{w}) = 1 \quad \Leftrightarrow \quad \Big(g(a,b-1,\overline{w}) = 1 \quad \vee \quad g(a,b-1,\overline{w} - w_j) = 1 \Big), \tag{4.15}$$

and

$$g(a,b,\overline{w}) = 1 \quad \Leftrightarrow \quad \Big(g(a+1,b,\overline{w}) = 1 \quad \vee \quad g(a+1,b,\overline{w}-w_j) = 1\Big). \quad (4.16)$$

In each step we maintain the sum $\overline{c} = \sum_{j=a}^{b} w_j$ and notice that we only need to store $g(a,b,\overline{w})$ for $0 \leq \overline{w} \leq \overline{c}$. The two recursions are run for $(a,b) := (s,s-1),(s,s),(s-1,s),(s-1,s+1)$ etc.

Now, using the techniques described in Section 4.1.2 we notice that the overall running time will be $O(nc/\log c)$ with corresponding space complexity $O(c/\log c)$. The benefit of the algorithm is, that we may stop the algorithm as soon as a solution to the decision problem has been found, i.e. when $g(a,b,c-\sum_{j=1}^{a-1} w_j) = 1$. We will denote the resulting algorithm WordsubsumPD.

4.1.4 Horowitz and Sahni Decomposition

Since (SSP) is a special case of (KP) it is straightforward to adapt the Horowitz and Sahni decomposition scheme described in Section 3.3 to the subset sum problem.

Ahrens and Finke [6] proposed an algorithm where the decomposition technique is combined with a branch-and-bound algorithm to reduce the space requirements. Furthermore a *replacement technique* (Knuth [279]) is used in order to combine the dynamic programming lists obtained by partitioning the variables into four subsets. The space bound of the Ahrens and Finke algorithm is $O(2^{n/4})$, making it best-suited for small but difficult instances.

Pisinger [386] presented a modified version of the Horowitz and Sahni decomposition, named Decomp, which combines good worst-case properties with quick solution times for easy problems. Let $z_b^{\mathrm{B}}(d)$ for $b = s-1,\ldots,n$ and $d = 0,\ldots,c$ be the optimal solution value to (SSP) restricted to variables $s,\ldots,b$ as follows:

$$z_b^{\mathrm{B}}(d) = \max\left\{ \sum_{j=s}^{b} w_j x_j \;\middle|\; \begin{array}{l} \sum_{j=s}^{b} w_j x_j \leq d, \\ x_j \in \{0,1\},\ j = s,\ldots,b \end{array} \right\}. \quad (4.17)$$

Let $z_a^{\mathrm{A}}(d)$ for $a = 1,\ldots,s$ and $d = 0,\ldots,c$ be the optimal solution value to (SSP) defined on variables $a,\ldots,s-1$ with the additional constraint $x_j = 1$ for $j = 1,\ldots,a-1$, thus

$$z_a^{\mathrm{A}}(d) = \max\left\{ \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{s-1} w_j x_j \;\middle|\; \begin{array}{l} \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{s-1} w_j x_j \leq d, \\ x_j \in \{0,1\},\ j = a,\ldots,s-1 \end{array} \right\}. \quad (4.18)$$

The recursion for z_b^{B} will repeatedly insert an item into the knapsack while the recursion for z_a^{A} will remove one item. Formally, the recursion z_b^{B} for $b = s,\ldots,n$ can be written as

$$z_b^{\mathrm{B}}(d) = \begin{cases} z_{b-1}^{\mathrm{B}}(d) & \text{if } d < w_b - 1, \\ \max\{z_{b-1}^{\mathrm{B}}(d),\ z_{b-1}^{\mathrm{B}}(d-w_b) + w_b\} & \text{if } d \geq w_b, \end{cases} \quad (4.19)$$

where $z^{\mathrm{B}}_{s-1}(d) = 0$ for $d = 0,\ldots,c$. The corresponding recursion z^{A}_a for $a = s-1, s-2,\ldots,1$ becomes

$$z^{\mathrm{A}}_a(d) = \begin{cases} z^{\mathrm{A}}_{a+1}(d) & \text{if } d > c - w_a, \\ \max\{z^{\mathrm{A}}_{a+1}(d),\ z^{\mathrm{A}}_{a+1}(d+w_a) - w_a\} & \text{if } d \le c - w_a, \end{cases} \tag{4.20}$$

with initial values $z^{\mathrm{A}}_s(d) = 0$ for $d \neq \hat{w}$ and $z^{\mathrm{A}}_s(\hat{w}) = \hat{w}$.

Now starting from $(a,b) = (s,s-1)$ the Decomp algorithm repeatedly uses the above two recursions, each time decreasing a and increasing b. At each iteration the two sets are merged in $O(c)$ time, in order to find the best current solution

$$z = \max_{d=0,\ldots,c} z^{\mathrm{B}}_b(d) + z^{\mathrm{A}}_a(c-d), \tag{4.21}$$

and the process is terminated if $z = c$, or $(a,b) = (1,n)$ has been reached.

The algorithm may be improved in those cases where $v := \sum_{j=1}^{n} w_j - \hat{w} < c$. Since there is no need for removing more items $j < s$ than can be compensated for by inserting items $j \ge s$, the recursion z^{A}_a may be restricted to consider capacities $d = c-v,\ldots,c$ while recursion z^{B}_b only will consider capacities $d = 0,\ldots,v$.

The Decomp algorithm has basically the same complexity $O(nc)$ as Bellman but if DP-with-Lists is used as described in Section 3.4, only undominated states need to be saved thus giving the complexity $O(\min\{nc,\ nv,\ 2^s + 2^{n-s}\})$. If c is of moderate size and hence only few items fit into the knapsack, then the resulting running time $O(nc)$ is attractive. If, on the other hand, the majority of the items fit into the knapsack, v becomes small and the running time $O(nv)$ is attractive. In the worst case, if c is huge and about half of the items fit into the knapsack, i.e. $s \approx n/2$, we get the time complexity $O(\sqrt{2^n})$.

4.1.5 Balancing

All the above dynamic programming recursions have the worst-case time complexity $O(nc)$ which corresponds to the complexity of the trivial Bellman recursion for the knapsack problem. For several years it was an open problem whether a more efficient recursion could be derived for the subset sum problem. Pisinger [387] answered this question in an affirmative way by presenting a recursion which runs in time $O(nw_{\max})$, hence dominating the complexity $O(nc)$ under the trivial assumption that $w_{\max} < c$.

In Section 3.6 we were introduced to the main principle of the balanced dynamic programming recursion. However, the algorithm was based on a recursive search procedure with a pool of open nodes P, storing intermediate results to improve the time complexity. To avoid the overhead of the recursion and memorization, we will here present an iterative version of the algorithm.

The table $g(b,\overline{w})$ for $b = s-1,\ldots,n$ and $\overline{w} = c - w_{\max} + 1,\ldots,c + w_{\max}$ is defined as in (3.4), i.e., we have

Algorithm Balsub:

```
1  for w̄ := c − w_max + 1 to c do g(s − 1, w̄) := 0
2  for w̄ := c + 1 to c + w_max do g(s − 1, w̄) := 1
3  g(s − 1, ŵ) := s
4  for b := s to n do
5     for w̄ := c − w_max + 1 to c + w_max do g(b, w̄) := g(b − 1, w̄)
6     for w̄ := c − w_max + 1 to c do
7        w̄' := w̄ + w_b, g(b, w̄') := max{g(b, w̄'), g(b − 1, w̄)}
8     for w̄ := c + w_b downto c + 1 do
9        for j := g(b − 1, w̄) to g(b, w̄) − 1 do
10          w̄' := w̄ − w_j, g(b, w̄') := max{g(b, w̄'), j}
```

Fig. 4.4. Algorithm Balsub is an iterative implementation of the balanced dynamic programming recursion.

s

j	1	2	3	4	5	6
w_j	6	4	2	6	4	3

$c = 15$

$\overline{w}$	$g(3,\overline{w})$	$g(4,\overline{w})$	$g(5,\overline{w})$	$g(6,\overline{w})$
10	0	1	1	1
11	0	0	0	1
12	4	4	4	4
13	0	0	0	2
14	0	2	3	3
15	0	0	0	4
16	1	3	4	4
17	1	1	1	3
18	1	4	4	4
19	1	1	1	1
20	1	1	1	1
21	1	1	1	1

Fig. 4.5. An instance of (SSP) and the corresponding table $g(b,\overline{w})$ as generated by algorithm Balsub. The split item is $s = 4$ and $w_{\max} = 6$ meaning that the table is defined for $\overline{w} = c - w_{\max} + 1, \ldots, c + w_{\max} = 10, \ldots, 21$. The optimal solution value is 15 meaning that the weight sum $\overline{w} = 15$ can be obtained from the split solution through balanced operations on items $4, \ldots, 6$ only.

$$g(b,\overline{w}) := \max_{a=1,\ldots,b+1} \left\{ a \;\middle|\; \begin{array}{l} \text{a balanced filling } x \text{ exists with} \\ \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{b} w_j x_j = \overline{w}, \\ x_j \in \{0,1\},\ j = a, \ldots, b \end{array} \right\} \tag{4.22}$$

where $g(b,\overline{w}) = 0$ if no filling exists.

Following the steps a) to f) described in Section 3.6 we get a very simple algorithm as illustrated in Figure 4.4. Using the idea from step f) we also use the table $g(b,\overline{w})$ to save open nodes in the search tree. Hence, if $a = g(b,\overline{w}) > 0$ then we have an open

node $(a,b,\overline{w})$ in the pool P at Figure 3.9. Due to the concept of dominance described in step c), only one open node will exist for each value of b and $\overline{w}$. When running through lines 4–11 in algorithm Balsub open nodes are found in table $g(b',\overline{w})$ for values of $b' \geq b$ where b is the current value in line 4.

In lines 1–3 we initialize table $g(b,\overline{w})$ as described in step a). In line 4 and further on we repeatedly consider the addition of the next item b. Line 5 corresponds to the case where we do not add item b and hence table $g(b-1,\overline{w})$ is simply copied to $g(b,\overline{w})$. Lines 6–7 correspond to the addition of item b, where the new weight sum becomes $\overline{w}' = \overline{w} + w_b$. Due to the balancing, item b can only be added to nodes with weight sum $\overline{w} \leq c$. The generated node is saved in table $g(b,\overline{w})$, where we test for dominance by choosing $g(b,\overline{w}') = \max\{g(b,\overline{w}'),\ g(b-1,\overline{w})\}$.

Having considered the insertion or omission of item b, lines 8–11 finally deal with the removal of one or more items located before $g(b,\overline{w})$. As it may be necessary to remove several items in order to maintain feasibility of the solution, we consider the entries for decreasing values of $\overline{w}$, thus allowing for several removals. An example of the Balsub algorithm is given in Figure 4.5.

An optimal solution value z^* is found as the largest value $\overline{w} \leq c$ such that $g(n,\overline{w}) \neq 0$. In order to determine the corresponding solution vector x^* we extend the fields in table g with a pointer to the "parent" entry. In this way we may follow back the entries in table g and hence determine those balanced operations made to reach the optimal solution.

The time and space complexity of algorithm Balsub is the same as described in Section 3.6. Array $g(b,\overline{w})$ has size $(n-s+1)(2w_{\max})$ hence using $O(nw_{\max})$ space. As for the time complexity we notice that lines 2-3 demand $2w_{\max}$ operations. Line 6 is executed $2w_{\max}(n-s+1)$ times. Line 7 is executed $w_{\max}(n-s+1)$ times. Finally, for each $\overline{w} > c$, line 11 is executed $g(n,\overline{w}) \leq b$ times in all. Thus during the whole process, line 11 is executed at most $sw_{\max}$ times.

Corollary 4.1.4 *The complexity of algorithm* Balsub *is* $O(nw_{\max})$ *in time and space.*

If all weights w_j are bounded by a fixed constant the complexity of the Balknap algorithm becomes $O(n)$, i.e. linear time.

It is quite straightforward to reduce the space complexity of algorithm Balsub at the cost of an increased time complexity.

Corollary 4.1.5 *The (SSP) can be solved using* $O(w_{\max})$ *space and* $O(n^2 w_{\max})$ *time.*

Proof. Since the main loop in lines 4–10 of Balsub calculates $g(b,\cdot)$ based on $g(b,\cdot)$ and $g(b-1,\cdot)$ we only need to store two columns of the table. When the algorithm terminates with $g(b,\overline{w}) = a$ for some value of $\overline{w}$ we notice from the definition of (4.22) that item a cannot have been chosen (since otherwise we could increase a),

and also that all items $j = 1,\ldots,a-1$ were chosen. Hence we can fix the solution vector for these items and solve the remaining problem again. Since we will remove at least one item a in each iteration, we will not use more than n iterations in total. □

4.1.6 Bellman Recursion in Decision Form

Yanasse and Soma [492] presented a variant of the Bellman recursion (4.7) where the optimization form is replaced by a decision form. The table $h(j,\overline{w})$ for $j = 1,\ldots,n$ and $\overline{w} = 0,\ldots,c$ is defined as

$$h(j,\overline{w}) := \min_{a=1,\ldots,j} \left\{ a \left| \begin{array}{l} \text{a solution } x \text{ exists with} \\ \sum_{j=1}^{a} w_j x_j = \overline{w}, \\ x_j \in \{0,1\},\ j = 1,\ldots,a \end{array} \right. \right\}, \tag{4.23}$$

where we set $h(j,\overline{w}) = n+1$ if no solution exists. Notice that this corresponds to the invariant (4.22) when considering general solutions and not only balanced solutions. The associated recursion becomes

$$h(j,\overline{w}) = \left\{ \begin{array}{ll} \min\{j,\ h(j-1,\overline{w})\} & \text{if } \overline{w} \geq w_j \text{ and } h(j-1,\overline{w}-w_j) \leq n \\ h(j-1,\overline{w}) & \text{otherwise} \end{array} \right\}, \tag{4.24}$$

with initial values $h(0,\overline{w}) = n+1$ for all $\overline{w} = 1,\ldots,c$, and $h(0,0) = 1$. An optimal solution solution value is found as

$$z^* = \max_{\overline{w}=0,\ldots,c} \{\overline{w} | h(n,\overline{w}) \leq n\}. \tag{4.25}$$

The time consumption of the recursion is $O(nc)$. The space consumption of the algorithm is $O(n+c)$ as we only need table $h(j-1,\cdot)$ to calculate $h(j,\cdot)$. This also holds if we want to find the optimal solution vector x^* since $g(n,\cdot)$ directly contains the index of the last item added, and hence it is easy to backtrack through $g(n,\cdot)$ to find the optimal solution vector. Like in the previous dynamic programming algorithms, one may use DP-with-Lists as described in Section 3.4, so that only undominated states need to be saved.

4.2 Branch-and-Bound

Several branch-and-bound algorithms have been presented for the solution of (SSP). In those cases where many solutions to the decision problem SSP-DECISION exist, branch-and-bound algorithms may have excellent solution times. If no solutions to SSP-DECISION exist, dynamic programming algorithms may be a better alternative.

4.2.1 Upper Bounds

The simplest upper bound appears by LP-relaxation of (SSP). Since all items j have the same "efficiency" $e_j = 1$ no sorting is needed to solve the relaxation, and the optimal solution value is found as

$$U_1 := U_{\mathrm{LP}} = c. \tag{4.26}$$

This trivial bound is of little use in a branch-and-bound algorithm since it only allows the algorithm to terminate if a solution to the decision problem SSP-DECISION has been found. Otherwise the branch-and-bound algorithm may be forced into an almost complete enumeration.

To obtain a tighter bound we use minimum and maximum cardinality bounds as proposed by e.g. Balas [18], Padberg [365] and Wolsey [488]. Starting with the maximum cardinality bound, assume that the weights are ordered in increasing order and define k as

$$k = \min\left\{h \left| \sum_{j=1}^{h} w_j > c \right.\right\} - 1. \tag{4.27}$$

We may now impose the *maximum cardinality constraint* to (SSP) as

$$\sum_{j=1}^{n} x_j \leq k. \tag{4.28}$$

An obvious way of using this constraint is to choose the k largest items getting the weight sum $\tilde{w} := \sum_{j=n-k+1}^{n} w_j$. This leads to the bound

$$U_2 := \min\{c, \tilde{w}\}. \tag{4.29}$$

If we surrogate relax the cardinality constraint with the original capacity constraint, using multipliers μ and 1, where $\mu \geq 0$, we get the problem $S(\mathrm{SSP}, \mu)$ given by

$$\begin{aligned} S(\mathrm{SSP},\mu) \qquad & \text{maximize} \sum_{j=1}^{n} w_j x_j \\ & \text{subject to} \sum_{j=1}^{n} (w_j + \mu) x_j \leq c + \mu k, \\ & x_j \in \{0,1\}, \qquad\qquad j = 1,\ldots,n\,, \end{aligned} \tag{4.30}$$

which is a so-called "inverse strongly correlated knapsack problem" (see Section 5.5 for details). Similarly, Lagrangian relaxing the cardinality constraint leads to a so-called "strongly correlated knapsack problem". As we are interested in bounds which may be derived in polynomial time, an additional LP-relaxation $C(S(\mathrm{SSP}, \mu))$ is considered. To solve the latter problem, assume that the items are ordered according to decreasing efficiencies $w_j/(w_j + \mu)$, and let the split item associated with the ordering be defined by

$$s' = \min\left\{h \left| \sum_{j=1}^{h} w_j + \mu > c + k\mu \right.\right\}. \tag{4.31}$$

The continuous solution is then given by

$$z(C(S(\mathrm{SSP},\mu))) = \sum_{j=1}^{s'-1} w_j + \left(c + k\mu - \sum_{j=1}^{s'-1}(w_j+\mu)\right)\frac{w_{s'}}{w_{s'}+\mu}. \tag{4.32}$$

Notice, that the sorting according to decreasing efficiencies $w_j/(w_j+\mu)$ for any $\mu > 0$ corresponds to a sorting according to decreasing weights, hence $s' = k+1$, and $\sum_{j=1}^{s'-1} w_j = \tilde{w}$. This means in particular that

$$z(C(S(\mathrm{SSP},\mu))) = \tilde{w} + (c-\tilde{w})\frac{w_{s'}}{w_{s'}+\mu}. \tag{4.33}$$

Since the latter is a convex combination of c and $\tilde{w}$ the bound obtained this way cannot be tighter than U_2 given by (4.29).

If we instead apply Lagrangian relaxation to the weight constraint of (SSP) using multiplier $\lambda \geq 0$, and keep the cardinality constraint (4.28) we get the problem $L(\mathrm{SSP},\lambda)$

$$\begin{aligned} L(\mathrm{SSP},\lambda) \qquad & \text{maximize } \sum_{j=1}^{n} w_j x_j - \lambda\left(\sum_{j=1}^{n} w_j x_j - c\right) \\ & \text{subject to } \sum_{j=1}^{n} x_j \leq k, \\ & \qquad x_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned} \tag{4.34}$$

The objective function may be written as

$$(1-\lambda)\sum_{j=1}^{n} w_j x_j + \lambda c. \tag{4.35}$$

For $\lambda \geq 1$ the optimal solution is found by not choosing any items, thus we get the value $z(L(\mathrm{SSP},\lambda)) = \lambda c \geq c$. For $0 \leq \lambda < 1$ the optimal value is found by choosing the $s-1$ largest values w_j. The objective value is again a convex combination of $\tilde{w}$ and c, and it will never be tighter than the bound (4.29).

We may conclude that none of the bounds $z(S(\mathrm{SSP},\lambda))$ and $z(L(\mathrm{SSP},\lambda))$ are tighter than the bound U_2. Similar, disappointing results are obtained for minimum cardinality bounds.

4.2.2 Hybrid Algorithms

Several hybrid algorithms have been proposed which combine the best properties of dynamic programming and branch-and-bound methods. The most successful hybrid

algorithms are those by Plateau and Elkihel [395] and algorithm MTS by Martello and Toth [330]. Both algorithms are conceptually very similar, hence we have only chosen to go into details with the latter as it is most widely known.

The MTS algorithm assumes that the weights are sorted in decreasing order

$$w_1 \geq w_2 \geq \ldots \geq w_n, \tag{4.36}$$

and applies dynamic programming to enumerate the solutions consisting of only the last (small) weights, while branch-and-bound is used to search through combinations of the first (large) weights. The motivation is that during branch-and-bound an extravagant expense is necessary to search for solutions which fill a small residual capacity. Thus by building a table of such solutions through dynamic programming, a kind of memorization is used to avoid solving overlapping subproblems.

The MTS algorithm actually builds two lists of partial solutions. First, the recursion (4.7) is used to enumerate items $\beta,\ldots,n$, for all weight sums not greater than c. Then the same recursion is used to enumerate items $\alpha,\ldots,n$, but only for weight sums up to a given constant $\bar{c}$. The constants $\alpha < \beta < n$ and $\bar{c} < c$ were experimentally determined as $\alpha = n - \min\{2\log w_1,\ 0.7n\}$, $\beta = n - \min\{2.5\log w_1,\ 0.8n\}$ and $\bar{c} = 1.3w_\beta$. The enumeration results in two tables defined as

$$\begin{aligned} z^\alpha(d) &= \max\left\{\sum_{j=\alpha}^{n} w_j x_j \;\middle|\; \begin{array}{l}\sum_{j=\alpha}^{n} w_j x_j \leq d,\\ x_j \in \{0,1\}, j=\alpha,\ldots,n\end{array}\right\}, \quad d=0,\ldots,\bar{c},\\ z^\beta(d) &= \max\left\{\sum_{j=\beta}^{n} w_j x_j \;\middle|\; \begin{array}{l}\sum_{j=\beta}^{n} w_j x_j \leq d,\\ x_j \in \{0,1\}, j=\beta,\ldots,n\end{array}\right\}, \quad d=0,\ldots,c. \end{aligned} \tag{4.37}$$

The branch-and-bound part of MTS repeatedly sets a variable x_j to 1 or 0, backtracking when either the current weight sum exceeds c or $j = \beta$. For each branching node, the dynamic programming lists are used to find the largest weight sum which fits into the residual capacity.

4.3 Core Algorithms

Most of the randomly generated instances considered in the literature have the property that numerous solutions to the decision problem SSP-DECISION exist. Since the solution process may be terminated as soon as the first solution is found, one may expect that only a relatively small subset of the items need to be explicitly considered when dealing with large-sized instances. This leads to the idea of a *core* of the problem, which we will cover in more details when dealing with (KP). A core for a subset sum problem is simply a sufficiently large subset of the items such that a solution to the decision problem SSP-DECISION can be found.

Different heuristic rules are used for choosing the core: A natural choice is to choose the items with the smallest weights, as they are generally easier to combine to match the capacity c. Another choice could be based on ensuring the largest possible diversity in the weights of the core.

4.3.1 Fixed Size Core

Martello and Toth [330] presented the first algorithm based on solving a *fixed size core problem*. The algorithm, denoted MTSL, may be outlined as follows:

For a given instance of (SSP) with no particular ordering of the items, let $s = \min\{h \mid \sum_{j=1}^{h} w_j > c\}$ be the split item, and define a core C as an appropriately chosen interval of items around the split item. Hence $C = \{s-\delta, \ldots, s+\delta\}$, where Martello and Toth experimentally found that $\delta = 45$ is sufficient for many instances appearing in the literature. With this choice of the core, the associated *core problem* becomes:

$$\begin{aligned} &\text{maximize} \sum_{j \in C} w_j x_j \\ &\text{subject to} \sum_{j \in C} w_j x_j \leq c - \sum_{j=1}^{s-\delta-1} w_j, \\ &\qquad x_j \in \{0,1\}, \quad j \in C. \end{aligned} \tag{4.38}$$

The above problem is solved using the MTS algorithm described in Section 4.2.2. If a solution is obtained which satisfies the weight constraint with equality, we may obtain an optimal solution to the main problem by setting $x_j = 1$ for $j < s-\delta$ and $x_j = 0$ for $j > s+\delta$. Otherwise δ is doubled, and the process is repeated.

4.3.2 Expanding Core

The dynamic programming algorithm based on recursions (4.12) and (4.13) may be seen as an expanding core algorithm, which gradually expands the core $C = \{a, \ldots, b\}$. No sorting and reduction of the items is necessary, hence we may simply alternate between the two recursions and terminate when a solution with value c is found. The resulting *expanding core algorithm* will be described in more details in Section 5.4.2.

Also the algorithm based on recursions (4.18) and (4.19) as an expanding core algorithm making use of Horowitz and Sahni decomposition. The resulting algorithm Decomp presented by Pisinger [386] only needs to enumerate a moderately small core $C = \{a, \ldots, b\}$ for most instances from the literature.

4.3.3 Fixed Size Core and Decomposition

Soma and Toth [441] presented a dynamic programming algorithm, called ST00, which combines Horowitz and Sahni decomposition for a small core with the Bellman recursion in decision form (4.24).

The core is chosen in $O(n)$ time as follows: Linear search is used to fill the knapsack, considering the items in the order they originally appear. When the split item s has been determined, the algorithm continues adding items which fit into the residual capacity. The selected items are then rearranged so that they have indices $1,\ldots,s'-1$, where s' is the split item of the rearranged problem. Items not selected obviously get indices $s',\ldots,n$. The core consists of the two sets $C_1 = \{s'-\delta,\ldots,s'-1\}$ and $C_2 = \{s',\ldots,s'-1+\delta\}$. The advantage of this approach should be that the weights in the core form a representative subset of the weights, which is not the case if some kind of sorting is used for selecting the core.

The core is solved by use of the recursion (4.24) for each set C_1 and C_2, running DP-with-Lists. Like in the Decomp algorithm described in Section 4.1.4, the two dynamic programming tables are merged at each iteration in order to find the best current solution (cf. equation (4.21)). If a solution $z = c$ has been found, the algorithm stops.

If no optimal solution has been found by complete enumeration of C_1 and C_2, the algorithm continues as follows: The dynamic programming table obtained by enumerating C_1 is embedded into a new dynamic programming algorithm based on the same recursion (4.24), but running for items $C_1 \cup (N \setminus (C_1 \cup C_2))$. The resulting dynamic programming table is merged with the dynamic programming table corresponding to C_2 in order to find the optimal solution z^*.

The ST00 algorithm is outlined in Figure 4.6. Choosing the two parts of the core of size $\delta = \log c$ each part of the core may be enumerated in $O(c)$ time since $2^{|C_i|} \leq c$ when $|C_i| = \log c$ for $i = 1,2$. The overall running time of the algorithm becomes $O(\max\{c(n-\log c^2), c\log c\})$. If $n < \log c$ this results in the running time $O(c\log c)$ which is larger than using the Bellman in time $O(nc)$. If $n \geq \log c$ we get the running time $O(nc - c\log c)$ which asymptotically is $O(nc)$. Although no improvement in the worst-case running time is obtained, the actual running time of several instances may be improved if an optimal solution is found by solving the core problem.

4.4 Computational Results: Exact Algorithms

We will compare all recent algorithms for the (SSP), including the MTSL, Decomp, Balsub, ST00, and Wordsubsum algorithm. A comparison between the Ahrens and Finke algorithm presented in Section 4.1.4 and MTSL can be found in the book by Martello and Toth [335].

Five types of data instances are considered:

Algorithm ST00:

1. Find the split item s using the greedy algorithm, rearranging selected items to the first positions.
2. Find cores C_1, C_2, and solve the associated core problems using recursion (4.18) for class C_1 and (4.19) for class C_2. In each iteration merge the two dynamic programming tables as in (4.21) to find the current solution z^ℓ.
3. If the found solution is $z^\ell = c$ stop
4. Run the Bellman recursion in decision form (4.24) for items $C_1 \cup (N \setminus (C_1 \cup C_2))$.
5. Merge the two dynamic programming tables to find the optimal solution z^*.

Fig. 4.6. The algorithm ST00 combines Horowitz-Sahni decomposition and the solution of a core problem.

- Problems pthree: w_j randomly distributed in $[1, 10^3]$, $c := \lfloor n10^3/4 \rfloor$.
- Problems psix: w_j randomly distributed in $[1, 10^6]$, $c := \lfloor n10^6/4 \rfloor$.
- Problems evenodd: w_j even, randomly distributed in $[1, 10^3]$, $c := 2\lfloor n10^3/8 \rfloor + 1$ (odd). Jeroslow [250] showed that every branch-and-bound algorithm using LP-based bounds enumerates an exponentially increasing number of nodes when solving evenodd problems.
- Problems avis: $w_j := n(n+1) + j$, and $c := n(n+1)\ \lfloor (n-1)/2 \rfloor + n(n-1)/2$. Avis [86] showed that any recursive algorithm which does not use dominance will perform poorly for the avis problems.
- Problems somatoth: These instances are due to Soma and Toth [441]. First, two weights w' and w'' are selected randomly in $[1, n]$, such that $\gcd(w', w'') = 1$ and such that $\frac{c}{w'} < \frac{n}{2}$ and $\frac{c}{w''} < \frac{n}{2}$. Then the weights w_j are generated as $w_j = \lceil \frac{j}{2} \rceil w'$ when j is even, and $w_j = \lceil \frac{j}{2} \rceil w''$ when j is odd. The capacity is chosen as $c := (w'-1)(w''-1) - 1$.

In the present section we will consider those algorithms which only solve the (SSP) for a single capacity, while the following section will deal with algorithms solving the all-capacities variant of the problem. All the computational experiments were carried out on a AMD ATHLON, 1.2 GHz with 768 Mb RAM.

The running times of five different algorithms are compared in Table 4.1: the branch-and-bound algorithm MTSL, the Decomp dynamic programming algorithm, the Balsub balanced dynamic programming algorithm, the hybrid ST00 algorithm, and finally the WordsubsumPD primal-dual word RAM algorithm. Each problem was tested with 100 instances, and a time limit of 10 hours was assigned for the solution of all problems in the series. A dash indicates that the 100 instances could not be solved within the time limit, or that the algorithm ran out of space.

algorithm	n	pthree	psix	evenodd	avis	somatoth
MTSL	10	0.0000	0.0000	0.0000	0.0000	0.0000
	30	0.0000	0.0006	0.3854	1.0716	0.0001
	100	0.0000	0.0006	—	—	1.6217
	300	0.0000	0.0007	—	—	—
	1000	0.0000	0.0006	—	—	—
	3000	0.0000	0.0006	—	□	—
	10000	0.0001	□	—	□	□
Decomp	10	0.0000	0.0000	0.0000	0.0000	0.0000
	30	0.0000	0.0000	0.0006	0.0001	0.0000
	100	0.0000	0.0001	0.0133	0.0181	0.0032
	300	0.0000	0.0001	0.2811	2.4454	0.2317
	1000	0.0000	0.0002	3.4533	354.0636	16.5620
	3000	0.0001	0.0002	31.0939	□	461.3043
	10000	0.0001	□	343.0673	□	□
Balsub	10	0.0001	0.5556	0.0001	0.0000	0.0000
	30	0.0001	0.9842	0.0003	0.0003	0.0001
	100	0.0000	0.4990	0.0010	0.0171	0.0063
	300	0.0000	0.3136	0.0026	1.7292	0.9256
	1000	0.0001	0.2784	0.0086	—	40.8362
	3000	0.0000	0.3237	0.0255	□	—
	10000	0.0001	□	0.0845	□	□
ST00	10	0.0000	0.0405	0.0000	0.0000	0.0000
	30	0.0008	0.9656	0.0004	0.0001	0.0000
	100	0.0003	0.0004	0.0066	0.0556	0.0021
	300	0.0002	0.0004	0.0517	1.7644	0.0381
	1000	0.0002	0.0003	0.5569	—	5.2155
	3000	0.0002	0.0004	4.9626	□	298.6002
	10000	0.0000	□	55.2049	□	□
WordsubsumPD	10	0.0000	0.0055	0.0000	0.0000	0.0000
	30	0.0000	0.0655	0.0000	0.0000	0.0000
	100	0.0000	0.0900	0.0002	0.0051	0.0001
	300	0.0000	0.0853	0.0023	1.6067	0.0012
	1000	0.0000	0.0898	0.0261	210.5487	0.0530
	3000	0.0000	0.0921	0.2795	□	6.2541
	10000	0.0000	□	5.3642	□	□

Table 4.1. Solution times in seconds (AMD ATHLON, 1.2 GHz). Note that WordsubsumPD in the present implementation does not find the optimal solution vector, but just the optimal solution value.

When comparing the algorithms, one should keep in mind that Balsub and WordsubsumPD are "clean" algorithms in the sense that they just implement the dynamic programming recursion. These algorithms can obviously be improved by combining them with additional techniques like: good heuristics, appropriate sorting of the weights, upper bounds, core of a problem, etc.

The problems pthree and psix have the property that several solutions to SSP-DECISION exist when n is large, thus the algorithms may terminate as soon as an optimal solution has been found. It is seen that all the algorithms perform well for

these instances, and in particular the MTSL, Decomp and ST00 algorithms have very low solution times.

For the evenodd problems no solutions to SSP-DECISION do exist, meaning that the algorithms are forced to a complete enumeration. For these instances, the Balsub algorithm has time complexity $O(n)$ since the weights are bounded by a constant, and also in practice it has the most promising solution times. The other algorithms perform reasonably well, although the MTSL branch-and-bound algorithm is not able to solve large instances.

For the avis and somatoth we observe a similar behavior. The dynamic programming algorithms solve these instances in reasonable time, while the branch-and-bound algorithm is not able to solve more than small instances. In particular the WordsubsumPD has a nice performance, although all dynamic programming algorithms roughly have the same profile.

The comparisons do not show a clear winner, since all algorithms have different properties. Both Decomp and MTSL are good algorithms for the randomly generated instances pthree and psix, which have many optimal solutions. For the most difficult instances, the theoretical worst-case complexity is the best indication of how the algorithms will perform.

4.4.1 Solution of All-Capacities Problems

We will now compare algorithms dealing with the all-capacities problem as introduced in Section 1.3. Only the Bellman recursion (4.7) and the Wordsubsum algorithms have the property of returning all sub-solutions. Two versions of the Bellman recursion have been implemented. The Belltab implementation is based on the straightforward recursion (4.7) using the principles from Section 2.3 to reduce the space consumption to $O(n+c)$. The Bellstate implementation is also based on the recursion (4.7) but using DP-with-Lists as described in Section 3.4. In the latter version only entries which correspond to a weight sum which actually can be obtained are saved, thus the algorithm is more effective for "sparse" problems (i.e. problems where quite few weight sums can be obtained). The worst-case space complexity of Bellstate is however $O(n+c)$. The basic algorithm for solving the recursion was taken from [386]. None of the considered algorithms determine a solution vector x^* corresponding to the optimal solution value z^*.

The word size of the processor is $W = 32$. The running times of the three algorithms are compared in Table 4.2. Entries marked with a $\square$ could not be generated as the capacity c does not fit within a 32-bit word, while entries marked with a dash could not be solved due to insufficient memory or time.

As expected, the word encoding leads to a considerably more efficient algorithm which in general is at least one order of magnitude faster than the two other implementations. For medium sized problems of pthree and evenodd we even observe an improvement of 200–400 times. This may seem surprising as the word size is

algorithm	n	pthree	psix	evenodd	avis	somatoth
Belltab	10	0.0000	0.3836	0.0000	0.0000	0.0000
	30	0.0005	3.9784	0.0005	0.0008	0.0000
	100	0.0067	44.0179	0.0067	0.7174	0.0009
	300	0.1335	395.2377	0.1089	79.0562	0.0326
	1000	3.2165	—	3.2583	—	6.0427
	3000	29.1376	—	29.2874	□	174.2857
	10000	323.8119	□	324.3007	□	□
Bellstate	10	0.0000	0.0000	0.0000	0.0000	0.0000
	30	0.0010	1.7560	0.0005	0.0001	0.0000
	100	0.0166	70.4943	0.0083	0.0830	0.0019
	300	0.5756	—	0.2248	8.5402	0.0701
	1000	6.8077	—	3.3274	—	10.2399
	3000	61.7440	—	30.7996	□	295.0788
	10000	—	□	349.2680	□	□
Wordsubsum	10	0.0000	0.0051	0.0000	0.0000	0.0000
	30	0.0000	0.0869	0.0000	0.0000	0.0000
	100	0.0003	1.0109	0.0003	0.0057	0.0000
	300	0.0026	9.1827	0.0026	1.6120	0.0013
	1000	0.0284	101.9758	0.0286	208.1471	0.0555
	3000	0.2893	919.6537	0.2898	□	6.2646
	10000	4.5516	□	5.2573	□	□

Table 4.2. Solution times in seconds as average over 100 instances (AMD ATHLON, 1.2 GHz).

$W = 32$, but due to the decreased space complexity caching gets improved, which results in an additional speedup. For large problems, which do not fit within the cache, the speedup decreases.

It is also interesting to observe the fundamental difference between the two data structures used for the Bellman recursion. For "sparse" problems like the avis instances it is more efficient to use DP-with-Lists while the table representation is more efficient for problems where nearly all possible weight sums can be achieved. The Wordsubsum algorithm is based on a table representation, and thus it performs best for the "dense" problems. But even for the "sparse" problems it is able to outperform the state representation by almost one order of magnitude, and Wordsubsum is able to solve larger problems due to the reduced space complexity.

4.5 Polynomial Time Approximation Schemes for Subset Sum

A straightforward approach for an approximation algorithm for (SSP) would be to transfer the algorithm Ext-Greedy for (KP) from Section 2.5. Recall that Ext-Greedy chooses among the solution value of the greedy solution and the item with maximum weight as alternative solution. No sorting is necessary, because for (SSP) the efficiency is equal to one for all items. This means that Greedy examines the items in any order and inserts each new into the knapsack if it fits. Consequently, it

runs in linear time. As for (KP) also Ext-Greedy has a tight worst-case performance guarantee of 1/2. The tightness can be seen from the following instance with three items: Let M be odd and suitably large. Set $c = M$, $w_1 = \frac{M+1}{2}$ and $w_2 = w_3 = \frac{M-1}{2}$.

As for (KP) the worst-case performance of Greedy can be arbitrarily bad. This will be improved if the items are sorted in decreasing order according to (4.36). The resulting algorithm has again a tight worst-case performance of 1/2 since Ext-Greedy compares the largest item with the solution of Greedy and now the largest item will always be packed first. Heuristics for (SSP) with better worst-case performance than 1/2 can be relatively easily obtained. Martello and Toth [331] run the Greedy n times on the item sets $\{1,\ldots,n\},\{2,\ldots,n\},\ldots,\{n\}$ which gives a 3/4-approximation algorithm in time of $O(n^2)$ again assuming a sorting of the items in decreasing order. Two linear time approximation algorithms with performance guarantees 3/4 and 4/5, respectively, have been proposed in Kellerer, Mansini and Speranza [265]. The two heuristics select the "large" items (items greater than $3/4\,c$ and greater than $4/5\,c$, respectively) according to the number of large items in an optimal solution and assign the other items in a greedy way. All these algorithms are outperformed by the *FPTAS* in Section 4.6.

As (SSP) is a special case of (KP), all the approximation algorithms which will be presented in Section 6.1 are suited for the application to (SSP) as well. Analogously to 6.1 the *PTAS* follow the idea of "guessing" a certain set of items included in the optimal solution by going through (all) possible candidate sets, and then filling the remaining capacity in some greedy way. Again, all these algorithms do not require any relevant additional storage and hence can be performed within $O(n)$ space.

More specifically, a *PTAS* usually splits the set of items into two subsets. Let $\varepsilon \in (0,1)$ be the accuracy required. Then the set of *small* items S contains the items whose weight does not exceed εc, whereas the set of *large* items L contains the remaining ones, whose weight is larger than εc. It is straightforward to see that an accuracy of ε in the solution of the sub-instance defined by the items in L is sufficient to guarantee the same accuracy for the overall instance. This is formulated in a slightly more general way in Lemma 4.5.1.

Lemma 4.5.1 *Let H be a heuristic for (SSP) which assigns for given ε first the large items to the knapsack and then inserts the small items into the knapsack by a greedy algorithm. Let z_L denote the solution value of H for the large items and z_L^* the optimal solution for L, respectively. If $z_L \geq (1-\tilde{\varepsilon})z_L^*$ for $\tilde{\varepsilon} > 0$, then*

$$z^H \geq (1-\tilde{\varepsilon})z^* \quad \text{or} \quad z^H \geq (1-\varepsilon)c$$

holds. In particular, H is a $(1-\varepsilon)$-approximation algorithm for $\tilde{\varepsilon} \leq \varepsilon$.

Proof. There are only two possibilities: Either there is a small item which is not chosen by the greedy algorithm or not. But the former can only happen if the current solution is already greater than $(1-\varepsilon)c$. In the latter case all small items are assigned to the knapsack and we have

$$z^H \geq (1-\tilde{\varepsilon})z_L + w(S) \geq (1-\tilde{\varepsilon})z_L^* + (1-\tilde{\varepsilon})w(S) \geq (1-\tilde{\varepsilon})z^*.$$

□

The first *PTAS* for (SSP) is due to Johnson [251]. Initially, a set of large items with maximal weight not exceeding the capacity c is selected. This is done by checking all subsets of large items with at most $\lceil 1/\varepsilon \rceil - 1$ elements, exploiting the fact that no more than $\lceil 1/\varepsilon \rceil - 1$ large items can be selected by a feasible solution. Then, the small items are assigned to the best solution by the greedy algorithm for the residual capacity. Consequently, the algorithm is running in $O(n^{\lceil 1/\varepsilon \rceil - 1})$ time. Lemma 4.5.1 with $\tilde{\varepsilon} = 0$ shows that the heuristic of Johnson is a $(1-\varepsilon)$-approximation algorithm which solves (SSP) optimally if the obtained solution value is less than or equal to $(1-\varepsilon)c$.

An improvement of Johnson's basic approach has been given by Martello and Toth [331]. Their *PTAS* is a generalization of their 3/4-approximation algorithm presented above and runs in $O(n^{\lceil \frac{2}{3} + \frac{1}{3\varepsilon} \rceil})$. A variation of Martello and Toth has been published by Soma et al. [442]. Their algorithm has the same running time but the authors announce it yields a better experimental behavior.

The best known *PTAS* for (SSP) requiring only linear storage was found by Fischetti [146], and can be briefly described as follows. For a given accuracy ε, the set of items is subdivided into small and large items as defined above. Furthermore, the set L of large items is partitioned into a *minimal* number of q buckets $B_1, \ldots, B_q$ such that the difference between the smallest and the biggest item in a bucket is at most εc. Clearly, $q \leq 1/\varepsilon - 1$. A q-tuple $(\nu_1, \ldots, \nu_q)$ is called *proper* if there exists a set $\tilde{L} \subseteq L$ with $|\tilde{L} \cap B_i| = \nu_i$, $i = 1, \ldots, q$ and $w(\tilde{L}) \leq c$. Then a procedure **Local** returns for any proper q-tuple a subset $\tilde{L} \subseteq L$ with $|\tilde{L} \cap B_i| = \nu_i$ $(i = 1, \ldots, q)$ and $w(\tilde{L}) \leq c$ such that $w(\tilde{L}) \geq (1-\tilde{\varepsilon})c$ or $w(\tilde{L}) \geq w(\bar{L})$ for each $\bar{L} \subseteq L$ with $|\bar{L} \cap B_i| = \nu_i$ $(i = 1, \ldots, q)$ and $w(\bar{L}) \leq c$. Procedure **Local** is completed by adding small items in a greedy way. Since the q-tuple which corresponds to an optimal solution is not generally known, all possible values of $\nu_1, \ldots, \nu_q$ are tried. Therefore, only the current best solution value produced by **Local** has to be stored. Any call to **Local** can be done in linear time and an upper bound for the number of proper q-tuples is given by $2\lceil \sqrt{1/\tilde{\varepsilon}} \rceil - 3$ which gives a running time bound of $O(n^{2\lceil \sqrt{1/\tilde{\varepsilon}} \rceil - 2})$, whereas the memory requirement is only $O(n)$, see [146]. In fact, a more careful analysis shows that the time complexity is also bounded by $O(n \log n + \phi(1/\varepsilon))$, where ϕ is a suitably-defined (exponentially growing) function. Moreover, a (simplified) variant of the scheme runs in $O(n + (1/\varepsilon)\log^2 1/\varepsilon + \phi(1/\varepsilon))$, but requires $O(n + 1/\varepsilon)$ space.

Of course, for reasonably small ε an *FPTAS* for (SSP) has much better time complexity than a *PTAS*. Moreover, the best *FPTAS* for (SSP) presented in Section 4.6 has only $O(n + 1/\varepsilon)$ memory requirement which is quite close to the $O(n)$ memory requirement of the known *PTAS*. So, we dispense with a more detailed description of the *PTAS* by Fischetti.

4.6 A Fully Polynomial Time Approximation Scheme for Subset Sum

The first *FPTAS* for (SSP) was suggested by Ibarra and Kim [241]. As usual, the items are partitioned into small and large items. The weights of the large items are scaled and then the problem with scaled weights and capacity is solved optimally through dynamic programming. The small items are added afterwards using the greedy algorithm. Their approach has time complexity $O(n \cdot 1/\varepsilon^2)$ and space complexity $O(n + 1/\varepsilon^3)$. Lawler [295] improved the scheme of Ibarra and Kim by a direct transfer of his scheme for the knapsack problem which uses a more efficient method of scaling. His algorithm has only $O(n + 1/\varepsilon^4)$ time and $O(n + 1/\varepsilon^3)$ memory requirement. Note that the special algorithm proposed in his paper for (SSP) does not work, since he makes the erroneous proposal to round up the item values.

Lawler claims in his paper that a combination of his approach (which is not correct) with a result by Karp [258] would give a running time of $O(n + 1/\varepsilon^2 \log(\frac{1}{\varepsilon}))$. Karp presents in [258] an algorithm for (SSP) with running time $n(\frac{1+\varepsilon}{\varepsilon}) \log_{1+\varepsilon} 2$ which is in $O(n \cdot 1/\varepsilon^2)$. Lawler states that replacing n by the number of large items $O((1/\varepsilon)\log(1/\varepsilon))$ would give a running time of $O(n + 1/\varepsilon^2 \log(\frac{1}{\varepsilon}))$. It can be easily checked that a factor of $\frac{1}{\varepsilon}$ is missing in the second term of the expression. Possibly, this mistake originates from the fact that there is a misprint in Karp's paper, giving a running time of $n(\frac{1+\varepsilon}{2}) \log_{1+\varepsilon} 2$ instead of the correct $n(\frac{1+\varepsilon}{\varepsilon}) \log_{1+\varepsilon} 2$.

The approach by Gens and Levner [167, 169] is based on a different idea. They use a dynamic programming procedure where at each iteration solution values are eliminated which differ from each other by at least a threshold value depending on ε. The corresponding solution set is then determined by standard backtracking. Their algorithm solves the (SSP) in $O(n \cdot 1/\varepsilon)$ time and space.

Gens and Levner [170] presented an improved *FPTAS* based on the same idea. The algorithm finds an approximate solution with relative error less than ε in time $O(\min\{n/\varepsilon, n + 1/\varepsilon^3\})$ and space $O(\min\{n/\varepsilon, n + 1/\varepsilon^2\})$.

The best *FPTAS* for (SSP) which we will describe in detail in the following, has been developed by Kellerer, Mansini, Pferschy and Speranza [268]. Their algorithm requires $O(\min\{n \cdot 1/\varepsilon, n + 1/\varepsilon^2 \log(1/\varepsilon)\})$ time and $O(n + 1/\varepsilon)$ space. A comparison of the listed algorithms is given in Table 4.3.

As the algorithm by Kellerer et al. which we will denote shortly as algorithm FPSSP, is rather involved, we will split the presentation in several parts.

A starting point for constructing a *FPTAS* for (SSP) would be to partition the items into small and large items, apply Bellman-with-Lists or another efficient dynamic programming procedure to the large items and then assign the small items by a greedy algorithm. This is also the principal procedure in algorithm FPSSP.

author	running time	space
Ibarra, Kim [241]	$O(n \cdot 1/\varepsilon^2)$	$O(n+1/\varepsilon^3)$
Lawler [295]	$O(n+1/\varepsilon^4)$	$O(n+1/\varepsilon^3)$
Gens, Levner [167, 169]	$O(n \cdot 1/\varepsilon)$	$O(n \cdot 1/\varepsilon)$
Gens, Levner [170]	$O(\min\{n/\varepsilon, n+1/\varepsilon^3\})$	$O(\min\{n/\varepsilon, n+1/\varepsilon^2\})$
Kellerer, Pferschy, Mansini, Speranza [268]	$O(\min\{n \cdot 1/\varepsilon, n+1/\varepsilon^2 \log(1/\varepsilon)\})$	$O(n+1/\varepsilon)$

Table 4.3. Fully polynomial approximation schemes for (SSP).

Instead of Bellman-with-Lists a much more sophisticated dynamic programming procedure is applied as depicted in Figure 4.7. Additionally, the set of large items is reduced by taking from each subinterval of item weights

$$I_j :=]j\varepsilon c, (j+1)\varepsilon c], \qquad j = 1, \ldots 1/\varepsilon - 1,$$

only the $\lceil \frac{1}{\varepsilon j} \rceil - 1$ smallest and $\lceil \frac{1}{\varepsilon j} \rceil - 1$ largest items. The selected items are collected in the so-called set of *relevant items* R and the other items are discarded (see Step 2 of algorithm FPSSP). Moreover, the accuracy ε is refined by setting

$$\varepsilon := \frac{1}{\lceil \frac{1}{\varepsilon} \rceil}, \tag{4.39}$$

such that after this modification $k := 1/\varepsilon$ becomes integer.

For the rest of this chapter let z^*_M denote the optimal solution value for any item set M which is subset of the ground set N. Furthermore, we denote with z^A the solution value for algorithm FPSSP.

Lemma 4.6.1 ensures that any optimal solution for the relevant items R is at most εc smaller than an optimal solution for the large items.

Lemma 4.6.1

$$z^*_R \geq (1-\varepsilon)c \quad or \quad z^*_R = z^*_L.$$

Proof. Denote by μ_j the number of items of L_j $(j = 1, \ldots, k-1)$ in an optimal solution for item set L. Since $\lceil \frac{k}{j} \rceil$ items of L_j have total weight strictly greater than $\lceil \frac{k}{j} \rceil j\varepsilon c \geq c$, there are at most $\lceil \frac{k}{j} \rceil - 1$ items of L_j in any feasible solution of (SSP) and $\lceil \frac{k}{j} \rceil - 1 \geq \mu_j$ follows. Hence, the set Ψ which consists of the μ_j smallest elements of L_j for all $j = 1, \ldots, k-1$ is a feasible solution set of (SSP) and a subset of the set of relevant items R.

Now we exchange iteratively items of Ψ which belong to the μ_j smallest elements of some set L_j with one of the μ_j biggest items of L_j. We finish this procedure either

Algorithm FPSSP:

1. Initialization and Partition into Intervals

modify ε according to (4.39) and set $k := \lceil \frac{1}{\varepsilon} \rceil$
let $S := \{j \mid w_j \leq \varepsilon c\}$ be the set of *small items*
let $L := \{j \mid w_j > \varepsilon c\}$ be the set of *large items*
for $j = 1, \ldots, 1/\varepsilon - 1$ do partition $]\varepsilon c, c]$ into $1/\varepsilon - 1$ intervals $I_j :=]j\varepsilon c, (j+1)\varepsilon c]$
denote the items in I_j by

$$L_j := \{i \mid j\varepsilon c < w_i \leq (j+1)\varepsilon c\}$$

set $n_j := |L_j|$
end for j

2. Reduction of the Large Items

for $j = 1, \ldots, k-1$ do
if $n_j > 2(\lceil \frac{k}{j} \rceil - 1)$ then
let K_j consist of the $\lceil \frac{k}{j} \rceil - 1$ smallest and
the $\lceil \frac{k}{j} \rceil - 1$ biggest items in L_j
else let K_j consist of all items in L_j
end for j
define the set of *relevant items* R by $R := \bigcup_{j=1}^{k-1} K_j$
set $\rho := |R|$

3. Dynamic Programming Recursion

perform a dynamic programming procedure $DP(R, c)$ on the set R
return solution value z_L and solution set X_L

4. Assignment of the Small Items

assign the items of S to a knapsack with capacity $c - z_L$ by the greedy algorithm
let z_S be the greedy solution value and X_S be the corresponding solution set
return $z^A := z_L + z_S$ and $X^A := X_L \cup X_S$

Fig. 4.7. Principal description of algorithm FPSSP for (SSP).

when $(1-\varepsilon)c \leq w(\Psi) \leq c$ or when Ψ collects the μ_j biggest items of L_j for all $j = 1, \ldots, k-1$. This is possible since the weight difference in each exchange step does not exceed εc. At the end Ψ is still a subset of R and either $w(\Psi) \geq (1-\varepsilon)c$ or Ψ consists of the μ_j biggest items of each interval and therefore $w(\Psi) \geq z_L^*$. □

Selecting the $\left\lceil \frac{k}{j} \right\rceil - 1$ items with smallest and largest weight for L_j $(j = 1, \ldots, k-1)$ in Step 2 can be done efficiently as described in Dor and Zwick [112] and takes altogether $O(n+k)$ time. The total number ρ of relevant items in R is bounded from

above by $\rho \leq n$ and by

$$\rho \leq 2\sum_{j=1}^{k-1}\left(\left\lceil\frac{k}{j}\right\rceil - 1\right) \leq 2k\sum_{j=1}^{k-1}\frac{1}{j} < 2k\log k.$$

Therefore,

$$\rho \leq O\left(\min\{n, \frac{1}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\}\right). \tag{4.40}$$

Consequently, the corresponding modification of Bellman-with-Lists requires running time $O(\min\{nc, n+1/\varepsilon\log(1/\varepsilon)c\})$ and space $O(n+(1/\varepsilon)c)$ but approximates the optimal solution still with accuracy εc. (For each partial sum we have to store at most $1/\varepsilon$ large items.) Note that Improved-Bellman has only $O(n+c)$ memory requirement.

The next step is to get an approximation algorithm with time and space complexity not depending on c, i.e. to find an appropriate dynamic programming procedure for Step 3 of algorithm FPSSP. A standard scaling of the set $\{0,1,\ldots,c\}$ of reachable values is not reasonable for (SSP). If we identify each interval $I_j =]j\varepsilon c, (j+1)\varepsilon c]$ with its lower bound $j\varepsilon c$ and apply Bellman, we have no guarantee for the feasibility of the obtained solution. On the other hand, taking the upper bound, can have the consequence that the obtained solution is arbitrarily bad, as the instance with the two items $1/2-\varepsilon/2$ and $1/2+\varepsilon/2$ shows. For this reason the smallest value $\delta^-(j)$ and the largest value $\delta^+(j)$ in each subinterval I_j are kept in each iteration and are updated if in a later recursion smaller or larger values in I_j are obtained. In principle we have replaced the c possible reachable values by $\frac{1}{\varepsilon}$ *reachable intervals* and perform instead of Bellman a so-called *"relaxed" dynamic programming*. This procedure Relaxed-Dynamic-Programming returns the array

$$\Delta = \{\delta[1] \leq \delta[2] \leq \ldots \leq \delta[k']\} \tag{4.41}$$

of *reduced reachable values*, i.e. Δ consists of the values $\delta^-(j), \delta^+(j)$ sorted in increasing order.

Relaxed-Dynamic-Programming, as depicted in Figure 4.8, is formulated not only for parameters R and c but also for arbitrary $\tilde{R} \subseteq R$ and $\tilde{c} \leq c$. The array Δ from (4.41) is identical to array $\Delta_{\tilde{R}}$. Additionally, the value $d(\delta)$ represents the index of the last item which is used to compute the iteration value δ. It is stored for further use in procedure Backtracking-SSP which will be described later on. Note that the last interval $I_{\tilde{k}}$ contains only values smaller than or equal to $\tilde{c}$.

Now we prove that also the reduction of the complete dynamic programming scheme to the smaller sets $\delta^+(\cdot), \delta^-(\cdot)$ is not far away from an optimal scheme for any subset of items.

Procedure Relaxed-Dynamic-Programming($\tilde{R}, \tilde{c}$):

Initialization

let $\tilde{R} = \{v_1, v_2, \ldots, v_{\tilde{\rho}}\}$ and $\tilde{\rho} := |\tilde{R}|$
compute $\tilde{k}$ with $\tilde{c} \in I_{\tilde{k}}$
$\delta^-(j) := \delta^+(j) := 0 \quad j = 1, \ldots, \tilde{k}$

Forward Recursion

for $i := 1$ to $\tilde{\rho}$ do
  form the set $\Delta_i := \{\delta^+(j) + v_i \mid \delta^+(j) + v_i \leq \tilde{c}, j = 1, \ldots, \tilde{k}\} \cup$
  $\{\delta^-(j) + v_i \mid \delta^-(j) + v_i \leq \tilde{c}, j = 1, \ldots, \tilde{k}\} \cup \{v_i\}$
  for all $u \in \Delta_i$ do
    compute j with $u \in I_j$
    if $\delta^-(j) = 0$ then
      $\delta^-(j) := \delta^+(j) := u$ and $d(\delta^-(j)) := d(\delta^+(j)) := i$
    if $u < \delta^-(j)$ then $\delta^-(j) := u$ and $d(\delta^-(j)) := i$
    if $u > \delta^+(j)$ then $\delta^+(j) := u$ and $d(\delta^+(j)) := i$
  end for all u
end for i
return $\delta^-(\cdot), \delta^+(\cdot), d(\cdot)$

Fig. 4.8. Procedure Relaxed-Dynamic-Programming returns the reduced reachable values.

The difference between an optimal dynamic programming scheme for some $\tilde{R}, \tilde{c}$ and the reduced version in procedure Relaxed-Dynamic-Programming is the following: For each new item i from 1 to $\tilde{\rho}$ an optimal algorithm would compute the sets

$$\Delta_i^* := \{\delta + v_i \mid \delta + v_i \leq \tilde{c}, \delta \in \Delta_{i-1}^*\} \cup \Delta_{i-1}^*$$

with $\Delta_0^* := \{0\}$. Thus, Δ_i^* is the set of the values of all possible partial solutions using the first i items of set $\tilde{R}$.

For the procedure Relaxed-Dynamic-Programming we define by

$$D_i := \{\delta^-(j), \delta^+(j) \mid j = 1, \ldots, \tilde{k}\} \tag{4.42}$$

the reduced set of $2\tilde{k}$ solution values computed in iteration i. The elements of each D_i ($i = 1, \ldots, \tilde{\rho}$) are renumbered in increasing order such that $D_i = \{\delta_i[s] \mid s = 1, \ldots, 2\tilde{k}\}$ and

$$\delta_i[1] \leq \delta_i[2] \leq \ldots \leq \delta_i[2\tilde{k}],$$

to set aside 0-entries. After these preparations it can been shown [268].

Lemma 4.6.2 *For each $\delta^* \in \Delta_i^*$ there exists some index ℓ with*

$$\delta_i[\ell] \leq \delta^* \leq \delta_i[\ell + 1] \quad \textit{and} \quad \delta_i[\ell + 1] - \delta_i[\ell] \leq \varepsilon c \tag{4.43}$$

or even

$$\delta_i[2\tilde{k}] \geq \tilde{c} - \varepsilon c. \tag{4.44}$$

Proof. The statement is shown by induction on i. The assertion is trivially true for $i = 1$. Let us assume it is true for all iterations from 1 to $i-1$.

Let $\delta_i^* \in \Delta_i^*$ and $\delta_i^* \notin \Delta_{i-1}^*$. Then, $\delta_i^* = \delta + v_i$ for some $\delta \in \Delta_{i-1}^*$. If $\delta_{i'}[2\tilde{k}] \geq \tilde{c} - \varepsilon c$ for some $i' \in \{1, \ldots, i-1\}$, the claim follows immediately. Otherwise, we assume by the induction hypothesis that there are $\delta_{i-1}[\ell]$, $\delta_{i-1}[\ell+1]$ with

$$\delta_{i-1}[\ell] \leq \delta \leq \delta_{i-1}[\ell+1] \quad \text{and} \quad \delta_{i-1}[\ell+1] - \delta_{i-1}[\ell] \leq \varepsilon c.$$

Set $a := \delta_{i-1}[\ell] + v_i$ and $b := \delta_{i-1}[\ell+1] + v_i$. Of course, $b - a \leq \varepsilon c$ and

$$a \leq \delta_i^* \leq b.$$

Assume first that δ_i^* is in the interval $I_{\tilde{k}}$ containing $\tilde{c}$. If $b > \tilde{c}$ then $\tilde{c} \geq a > \tilde{c} - \varepsilon c$, while if $b \leq \tilde{c}$, we get $b \geq \tilde{c} - \varepsilon c$. Hence, at least one of the values a, b fulfills inequality (4.44).

Assume now that $\delta_i^* \in I_j$ with $j < \tilde{k}$. We distinguish three cases:

(i) $a \in I_j, b \in I_j$,
(ii) $a \in I_j, b \in I_{j+1}$,
(iii) $a \in I_{j-1}, b \in I_j$.

In the remainder of the proof the values $\delta^-(j)$ and $\delta^+(j)$ are taken from iteration i. In case (i) we get that both $\delta^-(j)$ and $\delta^+(j)$ are not equal to zero, and (4.43) follows. For case (ii) note that $\delta^-(j+1) \leq b$. If $a = \delta^-(j)$, then $\delta^-(j+1) - \delta^+(j) \leq b - a \leq \varepsilon c$. If on the other hand $a > \delta^-(j)$, then $\delta^+(j) \geq a$ and again $\delta^-(j+1) - \delta^+(j) \leq \varepsilon c$. Hence (4.43) follows. Case (iii) is analogous to case (ii) and we have shown that (4.43) or (4.44) hold for each $\ell \in \{1, \ldots, \tilde{\rho}\}$. □

Let $\delta_{\max} := \delta[k'] = \delta_{r\tilde{h}o}[2\tilde{k}]$ be the largest value from the array of reduced reachable values. Then we conclude.

Corollary 4.6.3 *Performing procedure* Relaxed-Dynamic-Programming *with inputs $\tilde{R}$ and $\tilde{c}$ yields*

$$\textit{either} \quad \tilde{c} - \varepsilon c \leq \delta_{\max} \leq \tilde{c} \quad \textit{or} \quad \delta_{\max} = z_{\tilde{R}}^*.$$

Proof. If we use the construction of Lemma 4.6.2 for $i = \tilde{\rho}$ and set $\delta^* = z_{\tilde{R}}^*$, the corollary follows. □

Lemma 4.6.2 and Corollary 4.6.3 show that value $\delta_{\max}$ is at least $(1-\varepsilon)c$ or is even equal to the optimal solution value. Consequently, algorithm FPSSP with procedure Relaxed-Dynamic-Programming as dynamic programming recursion in Step 3, yields an $(1-\varepsilon)$-approximation algorithm. Replacing c by $\frac{1}{\varepsilon}$ it uses $O(n + 1/\varepsilon^2)$ space and has running time in $O(\min\{n/\varepsilon, 1/\varepsilon^2 \log(1/\varepsilon)\})$.

We have already achieved a FPTAS with the claimed running time, only the memory requirement is too large by a factor of $1/\varepsilon$ which is due to the fact that we store for each reduced reachable value i the corresponding solution set. Thus, if we would be satisfied with calculating only the maximal solution value and not be interested in the corresponding solution set, we could finish the algorithm after this step.

One way of reducing the space could be to store for each reduced reachable value $\delta[j]$ only the index $d(j)$ of the last item by which the reachable value was generated. Starting from the maximal solution value we then try to reconstruct the corresponding solution set by backtracking like in Improved-Bellman. But the partial sum (with value in I_i) which remains after each step of backtracking may not be stored anymore in Δ. So, if original values in I_i are no longer available we could choose one of the updated values $\delta^-(i)$, $\delta^+(i)$. Let y^B denote the total weight of the current solution set determined by backtracking. Lemma 4.6.4 will show in principle that there exists $y^{\hat{B}} \in \{\delta^-(i), \delta^+(i)\}$ with $(1-\varepsilon)c \leq y^{\hat{B}} + y^B \leq c$. Hence, we could continue backtracking with the stored value $y^{\hat{B}}$.

However, during backtracking another problem may occur: The series of indices, from which we construct our solution set, may increase after a while which means that an entry for I_i may have been updated after considering the item with index $d(j)$. This opens the unfortunate possibility that we take an item twice in the solution. Therefore, we can run procedure Backtracking-SSP only as long as the values $d(j)$ are decreasing. Then we have to recompute the remaining part of the solution by running again the relaxed dynamic programming procedure on a reduced item set $\tilde{R}$ consisting of all items from R with smaller index than the last value $d(j)$ and for the smaller subset capacity $\tilde{c} := c - y^B$. In the worst-case it may happen that Backtracking-SSP always stops after identifying only a single item of the solution set. This would increase the running time by a factor of $1/\varepsilon$.

Thus, to reconstruct the approximate solution set, Kellerer et al. [268] apply again the idea of the storage reduction scheme of Section 3.3.After performing Backtracking-SSP until the values $d(j)$ increase, procedure Divide-and-Conquer is called. Divide-and-Conquer is constructed similarly to procedure Recursion of Section 6.2.4. Procedures Backtracking-SSP, Divide-and-Conquer and the final version of Step 3 of algorithm FPSSP are depicted in Figures 4.9, 4.10 and 4.11. Note that both procedure Backtracking-SSP and procedure Divide-and-Conquer are formulated for arbitrary capacities $\tilde{c} \leq c$ and item sets $\tilde{R} \subseteq R$.

From Lemma 4.6.1 and Corollary 4.6.3 it follows immediately that the first execution of procedure Relaxed-Dynamic-Programming (performed in Step 3 of algorithm FPSSP) computes a solution value within εc of the optimal solution. In fact, if the optimal solution z_L^* is smaller than $(1-\varepsilon)c$, even the exact optimal solution is found. To preserve this property during the remaining part of the algorithm the computation with an artificially decreased capacity is continued although this would not be necessary to compute just an ε–approximate solution.

Step 3 of Algorithm FPSSP (final version):

$X_L := \emptyset$ *current solution set of large items*
$R^E := \emptyset$ *set of relevant items not further considered*
These two sets are updated only by procedure Backtracking-SSP.

call Relaxed-Dynamic-Programming(R, c)
returning $\delta^-(\cdot)$,$\delta^+(\cdot)$ and $d(\cdot)$
if $\delta_{\max} < (1-\varepsilon)c$ then set $c := \delta_{\max} + \varepsilon c$

perform Backtracking-SSP $(\delta^-(\cdot), \delta^+(\cdot), d(\cdot), R, c)$ returning y^B
if $c - y^B > \varepsilon c$ then
    call Divide-and-Conquer$(R \setminus R^E, c - y^B)$
    return y^{DC}
$z_L := y^B + y^{DC}$

Fig. 4.9. Final version of Step 3 of algorithm FPSSP.

Procedure Backtracking-SSP$(\delta^-(\cdot), \delta^+(\cdot), d(\cdot), \tilde{R}, \tilde{c})$:

Initialization
    $u := \max_j \{u'_j \mid u'_j = \delta^+(j) \text{ and } u'_j \le \tilde{c}\}$
    $y^B := 0$; *stop* := false

Backward Recursion
    repeat *items are added to* X_L *and deleted from* R^E
        $i := d(u)$
        $X_L := X_L \cup \{v_i\}$
        $y^B := y^B + v_i$
        $u := u - v_i$
        if $u > 0$ then
            compute j with $u \in I_j$
            if $\delta^+(j) + y^B \le \tilde{c}$ and $d(\delta^+(j)) < i$ then
                $u := \delta^+(j)$
            else if $\delta^-(j) + y^B \ge \tilde{c} - \varepsilon c$ and $d(\delta^-(j)) < i$ then
                $u := \delta^-(j)$
            else *stop* := true
        end if
    until $u = 0$ or *stop*
    $R^E := R^E \cup \{v_j \in \tilde{R} \mid j \ge i\}$
    return (y^B) *collected part of the solution value*

Fig. 4.10. Backtracking-SSP helps to recalculate the solution set.

Like Recursion, procedure Divide-and-Conquer splits this task into two subproblems by partitioning $\tilde{R}$ into two subsets R_1, R_2 of (almost) the same cardinality. Relaxed-Dynamic-Programming is then performed for both item sets independently with capacity $\tilde{c}$ returning two arrays of reduced reachable arrays Δ_1, Δ_2. Indeed, the

Procedure Divide-and-Conquer($\tilde{R}, \tilde{c}$)

Divide
  partition $\tilde{R}$ into two disjoint subsets R_1, R_2 with
  $|\tilde{R}_1| \approx |\tilde{R}_2| \approx |\tilde{L}|/2$
  call Relaxed-Dynamic-Programming($R_1, \tilde{c}$)
  returning $\delta_1^-(\cdot), \delta_1^+(\cdot), d_1(\cdot)$
  call Relaxed-Dynamic-Programming($R_2, \tilde{c}$)
  returning $\delta_2^-(\cdot), \delta_2^+(\cdot), d_2(\cdot)$

Conquer
  find entries u_1, u_2 of $\delta_1^-(\cdot), \delta_1^+(\cdot)$ and $\delta_2^-(\cdot), \delta_2^+(\cdot)$, respectively, with $u_1 \geq u_2$ such that

$$\tilde{c} - \varepsilon c \leq u_1 + u_2 \leq \tilde{c}$$

  $y_1^{DC} := 0; y_2^B := 0; y_2^{DC} := 0$ *local variables*

Resolve R_1
  call Backtracking-SSP($\delta_1^-(\cdot), \delta_1^+(\cdot), d_1(\cdot), R_1, \tilde{c} - u_2$)
  returning y_1^B
  if $\tilde{c} - u_2 - y_1^B > \varepsilon c$ then
    call Divide-and-Conquer($R_1 \setminus R^E, \tilde{c} - u_2 - y_1^B$)
    returning y_1^{DC}

Resolve R_2
  if $u_2 > 0$ then
    if $\tilde{c} - u_2 - y_1^B > \varepsilon c$ then
      call Relaxed-Dynamic-Programming($R_2, \tilde{c} - y_1^B - y_1^{DC}$)
      returning $\delta_2^-(\cdot), \delta_2^+(\cdot), d_2(\cdot)$. (*)
    call Backtracking-SSP($\delta_2^-(\cdot), \delta_2^+(\cdot), d_2(), R_2, \tilde{c} - y_1^B - y_1^{DC}$)
    returning y_2^B
    if $\tilde{c} - y_1^B - y_1^{DC} - y_2^B > \varepsilon c$ then
      call Divide-and-Conquer($R_2 \setminus R^E, \tilde{c} - y_1^B - y_1^{DC} - y_2^B$)
      returning y_2^{DC}
  end if
  $y^{DC} = y_1^B + y_1^{DC} + y_2^B + y_2^{DC}$
  return (y^{DC}) *part of the solution value contained in* $\tilde{R}$

Fig. 4.11. The recursive procedure Divide-and-Conquer.

situation with procedure Divide-and-Conquer is more complicated since in contrast to Recursion the entries $u_1 \in \Delta_1$, $u_2 \in \Delta_2$ do *not* necessarily sum up to $\tilde{c}$. It can be only shown that their sum is not far away from $\tilde{c}$.

To find the solution sets corresponding to values u_1 and u_2 first procedure Backtracking-SSP is executed for item set R_1 with capacity $\tilde{c} - u_2$ which reconstructs a part of the solution contributed by R_1 with value y_1^B. If y_1^B is not close enough to $\tilde{c} - u_2$ and hence does not fully represent the solution value generated by items in

R_1, a recursive execution of Divide-and-Conquer is performed for item set R_1 with capacity $\tilde{c} - u_2 - y_1^B$ which finally produces y_1^{DC} such that $y_1^B + y_1^{DC}$ is close to u_1.

The same strategy is carried out for R_2 producing a partial solution value y_2^B by Backtracking-SSP and possibly performing recursively Divide-and-Conquer which again returns a value y_2^{DC}. Note that the recomputation of $\delta_2^-(\cdot)$ and $\delta_2^+(\cdot)$ (see $(*)$ of Divide-and-Conquer) is necessary if the memory for $\delta_2^-(\cdot), \delta_2^+(\cdot), d_2(\cdot)$ was used during the recursive execution of procedure Divide-and-Conquer to resolve R_1. Alternatively, the storage structure could also be reorganized like in procedure Recursion defined in Section 6.2.4. All together the solution contributed by item set $\tilde{R}$ can be described as

$$y^{DC} = y_1^B + y_1^{DC} + y_2^B + y_2^{DC}.$$

Lemma 4.6.4 shows that there is always a subset of the remaining relevant items such that their total sum plus the collected partial solution value y^B is close to the capacity $\tilde{c}$.

Lemma 4.6.4 *After performing procedure* Backtracking-SSP *with inputs* $\delta^-(\cdot)$, $\delta^+(\cdot)$, $\tilde{R}$ *and* $\tilde{c}$ *we have* $y^B > \varepsilon c$ *and there exists a subset of items in* $\tilde{R} \setminus R^E$ *summing up to* $y^{\hat{B}}$ *such that*

$$\tilde{c} - \varepsilon c \leq y^B + y^{\hat{B}} \leq \tilde{c}. \tag{4.45}$$

Proof. In the first iteration one relevant item (with weight $> \varepsilon c$) is always added to y^B.

We will show that during the execution of Backtracking-SSP the value u always fulfills the properties required by $y^{\hat{B}}$ in every iteration.

Procedure Backtracking-SSP is always performed (almost) immediately after procedure Relaxed-Dynamic-Programming. Moreover, the value $\tilde{c}$ is either identical to the capacity in the preceding dynamic programming routine and hence with Corollary 4.6.3 the starting value of u fulfills (4.45) (with $y^{\hat{B}} = u$ and $y^B = 0$) or, if called while resolving R_1, u_1 has the same property.

During the computation in the loop we update u at first by $u - v_{d(u)}$. Hence, this new value must have been found during the dynamic programming routine while processing items with a smaller index than that leading to the old u. Therefore, we get for $u > 0$, that $\delta^-(j) \leq u \leq \delta^+(j)$. At this point $\tilde{c} - \varepsilon c \leq y^B + v_{d(u)} + u - v_{d(u)} \leq \tilde{c}$ still holds. Then there are two possible updates of u: We may set $u := \delta^+(j)$ thus not decreasing u and still fulfilling (4.45). We may also have $u := \delta^-(j)$ with the condition that $\delta^-(j) \geq \tilde{c} - y^B - \varepsilon c$ and hence inequality (4.45) still holds because u was less than $\tilde{c} - y^B$ and is further decreased.

Ending the loop in Backtracking-SSP there is either $u = 0$, and (4.45) is fulfilled with $y^{\hat{B}} = 0$, or *stop* = true, and by the above argument $y^{\hat{B}} := u$ yields the required properties.

At the end of each iteration (except possibly the last one) u is always set to an entry in the dynamic programming array which was reached by an item with index smaller than the item previously put into X_L. Naturally, all items leading to this entry must have had even smaller indices. Therefore, the final value $y^{\hat{B}}$ must be a combination of items with indices less than that of the last item added to X_L, i.e. from the set $\tilde{R} \setminus R^E$. □

In the next lemma it is shown that the sum of the values u_1, u_2 from Divide-and-Conquer are close to $\tilde{c}$.

Lemma 4.6.5 *If at the start of procedure* Divide-and-Conquer *there exists a subset of $\tilde{R}$ with weight $\tilde{y}$ such that*

$$\tilde{c} - \varepsilon c \leq \tilde{y} \leq \tilde{c},$$

then there exist u_1, u_2 fulfilling

$$\tilde{c} - \varepsilon c \leq u_1 + u_2 \leq \tilde{c}. \tag{4.46}$$

Proof. Obviously, we can write $\tilde{y} = y_1 + y_2$ with y_1 being the sum of items from R_1 and y_2 from R_2. If y_1 or y_2 is 0, the result follows immediately from Lemma 4.6.2 setting u_1 or u_2 equal to 0, respectively.

With Lemma 4.6.2 we conclude that after the Divide step there exist values a_1, b_1 from the dynamic programming arrays $\delta_1^-(\cdot), \delta_1^+(\cdot)$ with

$$a_1 \leq y_1 \leq b_1 \quad \text{and} \quad b_1 - a_1 \leq \varepsilon c.$$

Analogously, there exist a_2, b_2 from $\delta_2^-(\cdot), \delta_2^+(\cdot)$ with

$$a_2 \leq y_2 \leq b_2 \quad \text{and} \quad b_2 - a_2 \leq \varepsilon c.$$

Now it is easy to see that at least one of the four pairs from $\{a_1, b_1\} \times \{a_2, b_2\}$ fulfills (4.46). □

The return value of the procedure Divide-and-Conquer can be characterized in the following way.

Lemma 4.6.6 *If, performing procedure* Divide-and-Conquer *with inputs $\tilde{R}$ and $\tilde{c}$,*

$$\text{there exists a subset of } \tilde{R} \text{ with weight } \tilde{y} \text{ such that } \tilde{c} - \varepsilon c \leq \tilde{y} \leq \tilde{c}, \tag{4.47}$$

then also the returned value y^{DC} fulfills

$$\tilde{c} - \varepsilon c \leq y^{DC} \leq \tilde{c}.$$

Proof. Lemma 4.6.5 guarantees that under condition (4.47) there are always values u_1, u_2 satisfying (4.46).

Like in Section 3.2 the recursive structure of the Divide-and-Conquer calls can be seen as an ordered, not necessarily complete, binary rooted tree. Each node in the tree corresponds to one call of Divide-and-Conquer with the root indicating the first call from Step 3. Furthermore, every node may have up to two child nodes, the *left* child corresponding to a call of Divide-and-Conquer to resolve R_1 and the *right* child corresponding to a call generated while resolving R_2. As the left child is always visited first (if it exists), the recursive structure corresponds to a preordered tree walk.

In the following we will show the statement of the Lemma by backwards induction moving "upwards" in the tree, i.e. beginning with its leaves and applying induction to the inner nodes.

We start with the leaves of the tree, i.e. executions of Divide-and-Conquer with no further recursive calls. Therefore, we have $y_1^{DC} = 0$ after resolving R_1 and by considering the condition for not calling the recursion,

$$\tilde{c} - u_2 - \varepsilon c \leq y_1^B \leq \tilde{c} - u_2.$$

Resolving R_2 we either have $u_2 = 0$ and hence $y^{DC} = y_1^B$ and we are done with the previous inequality or we get

$$\tilde{c} - y_1^B - \varepsilon c \leq y_2^B \leq \tilde{c} - y_1^B$$

and hence with $y_2^{DC} = 0$

$$\tilde{c} - \varepsilon c \leq y_1^B + y_2^B = y^{DC} \leq \tilde{c}.$$

For all other nodes we show that the above implication is true for an arbitrary node under the inductive assumption that it is true for all its children. To do so, we will prove that if condition (4.47) holds, it is also fulfilled for any child of the node and hence by induction the child nodes return values according to the above implication. These values will be used to show that also the current node returns the desired y^{DC}.

If the node under consideration has a left child, we know by Lemma 4.6.4 that after performing procedure Backtracking-SSP with $\tilde{c} = \tilde{c} - u_2$, there exists $y_1^{\hat{B}}$ fulfilling

$$\tilde{c} - y_1^B - u_2 - \varepsilon c \leq y_1^{\hat{B}} \leq \tilde{c} - y_1^B - u_2$$

which is equivalent to the required condition (4.47) for the left child node. By induction, we get with the above statement for the return value of the left child (after rearranging)

$$\tilde{c} - u_2 - \varepsilon c \leq y_1^B + y_1^{DC} \leq \tilde{c} - u_2.$$

If there is no left child (i.e. for the case that $y_1^{DC} = 0$), we get from the condition for this event

$$\tilde{c} - u_2 - \varepsilon c \leq y_1^B \leq \tilde{c} - u_2.$$

If $u_2 = 0$ and hence $y^{DC} = y_1^B + y_1^{DC}$, we are done immediately in both of these two cases.

If there is a right child node we proceed along the same lines. From Lemma 4.6.4 we know that after performing Backtracking-SSP with $\tilde{c} = \tilde{c} - y_1^B - y_1^{DC}$ while resolving R_2 there exists $y_2^{\hat{B}}$ with

$$\tilde{c} - y_1^B - y_1^{DC} - y_2^B - \varepsilon c \leq y_2^{\hat{B}} \leq \tilde{c} - y_1^B - y_1^{DC} - y_2^B,$$

which is precisely condition (4.47) for the right child node.

Hence, by induction we can apply the above implication on the right child and get

$$\tilde{c} - y_1^B - y_1^{DC} - y_2^B - \varepsilon c \leq y_2^{DC} \leq \tilde{c} - y_1^B - y_1^{DC} - y_2^B$$

which is (after rearranging) equivalent to the desired statement for y^{DC} in the current node.

If there is no right child (i.e. $y_2^{DC} = 0$), the result follows from the corresponding condition. □

Applying this Lemma to the beginning of the recursion and considering also the small items Kellerer et al. [268] could finally state

Theorem 4.6.7 *Algorithm* FPSSP *is an* ε*–approximation scheme for (SSP). In particular*

$$z^A \geq (1-\varepsilon)c \quad \text{or} \quad z^A = z^*.$$

Moreover, the bound is tight.

Proof. As shown in Lemma 4.6.1 the reduction of the total item set to R in Step 2 does not eliminate all ε–approximate solutions. Hence, it is sufficient to show the claim for the set of relevant items, namely that at the end of Step 3 we have either

$$z_L \geq (1-\varepsilon)c \quad \text{or} \quad z_L = z_R^*.$$

If the first execution of Relaxed-Dynamic-Programming in Step 3 returns $\delta_{\max} < (1-\varepsilon)c$, we know from Corollary 4.6.3 that we have found the optimum solution value over the set of relevant items. Continuing with the updated capacity c (see Step 3 of FPSSP) the claim $z_L = z_R^*$ would follow immediately from the first alternative for the new capacity. Therefore, we will assume in the sequel that in Step 3 we find $\delta_{\max} \geq (1-\varepsilon)c$ and only prove that

$$z_L \geq (1-\varepsilon)c.$$

If Divide-and-Conquer is not performed at all, this relation follows immediately. It has to be shown that for y^{DC}, i.e. the return value of the first execution of Divide-and-Conquer,

$$(1-\varepsilon)c - y^B \leq y^{DC} \leq c - y^B$$

holds, because at the end of Step 3 we set $z_L := y^B + y^{DC}$. However, due to Lemma 4.6.4, the existence of a value $y^{\hat{B}}$ satisfying this relation is established. But this is exactly the condition required in Lemma 4.6.6 to guarantee that y^{DC} fulfills the above. The assertion follows now directly from Lemma 4.5.1.

To prove that the bound is tight, we consider the following series of instances: $\varepsilon = 1/t$, $n = 2t-1$, $c = tM$ with $M > t-1$ and $w_1 = \ldots = w_{t-1} = M+1$, $w_t = \ldots = w_{2t-1} = M$. It follows that $z^A = (t-1)M + (t-1)$ and $z^* = tM$. The performance ratio tends to $(t-1)/t$ when M tends to infinity. □

The asymptotic running time of algorithm FPSSP is analyzed in the following theorem.

Theorem 4.6.8 *For every accuracy $\varepsilon > 0$ $(0 < \varepsilon < 1)$ algorithm* FPSSP *runs in time $O(\min\{n \cdot 1/\varepsilon, n + 1/\varepsilon^2 \log(1/\varepsilon)\})$ and space $O(n + 1/\varepsilon)$. Especially, only $O(1/\varepsilon)$ storage locations are needed in addition to the description of the input itself.*

Proof. Throughout the algorithm, the relevant space requirement consists of storing the n items and six dynamic programming arrays $\delta_1^-(\cdot)$, $\delta_1^+(\cdot)$, $d_1(\cdot)$, $\delta_2^-(\cdot)$, $\delta_2^+(\cdot)$ and $d_2(\cdot)$ with length k.

Special attention has to be paid to the implicit memory necessary for the recursion of procedure Divide-and-Conquer. To avoid using new memory for the dynamic programming array in every recursive call, we always use the same space for the six arrays. But this means that after returning from a recursive call while resolving R_1 the previous data in δ_2^- and δ_2^+ is lost and has to be recomputed.

A natural bipartition of each $\tilde{R}$ can be achieved by taking the first half and second half of the given sequence of items. This means that each subset R_i can be represented by the first and the last index of consecutively following items from the ground set. If for some reason a different partition scheme is desired, a labeling method can be used to associate a unique number with each call of Divide-and-Conquer and with all items belonging to the corresponding set $\tilde{R}$.

Therefore, each call to Divide-and-Conquer requires only a constant amount of memory and the recursion depth is bounded by $O(\log k)$. Hence, all computations can be performed within $O(n + 1/\varepsilon)$ space.

Step 1 requires $O(n+k)$ time. By inequality (4.40), the parameter ρ is of order $O(\min\{n, 1/\varepsilon \log(1/\varepsilon)\})$.

Each call of procedure Relaxed-Dynamic-Programming with parameters $\tilde{R}$ and $\tilde{c}$ takes $O(\frac{\tilde{\rho}\cdot\tilde{c}}{\varepsilon c})$ time, as for each item i, $i = 1,\ldots,\tilde{\rho}$, we have to consider only $|\Delta_i|$ candidates for updating the dynamic programming arrays, a number which is clearly in $O(\frac{1}{\varepsilon})$.

Backtracking-SSP always immediately follows procedure Relaxed-Dynamic-Programming and can clearly be done in $O(\tilde{c}/t)$ time.

Therefore, Step 3 requires $O(\min\{n\cdot 1/\varepsilon, 1/\varepsilon^2\log(1/\varepsilon)\})$ time, applying the above bound on ρ, plus the effort of the Divide-and-Conquer execution which will be treated below. Clearly, Step 4 requires $O(n)$ time.

To estimate the running time of the recursive procedure Divide-and-Conquer we recall the representation of the recursion as a binary tree as used in the proof of Lemma 4.6.6 or in the general scheme of Section 3.3. A node is said to have *level* ℓ if there are $\ell - 1$ nodes on the path to the root node. The root node is assigned level 0. This means that the level of a node gives its recursion depth and indicates how many bipartitionings of the item set took place to reach $\tilde{R}$ starting at the root with R. Naturally, the maximal level is $\log\rho$ which is in $O(\log k)$.

Obviously, for any node with level ℓ the number of items considered in the corresponding execution of procedure Divide-and-Conquer, is bounded by $\tilde{\rho} < \frac{\rho}{2^\ell}$.

Let us describe the computation time in a node which consists of the computational effort without the at most two recursive calls to Divide-and-Conquer. If the node under consideration corresponds to an execution of Divide-and-Conquer with parameters $\tilde{R}$ and $\tilde{c}$, then the two calls to Relaxed-Dynamic-Programming from Divide take $O(\tilde{\rho}\cdot\frac{\tilde{c}}{\varepsilon c})$ time (see above) and dominate the remaining computations. Therefore, the computation time in a node with level ℓ is in $O(\frac{\rho}{2^\ell}\cdot\frac{\tilde{c}}{\varepsilon c})$.

For every node with input capacity $\tilde{c}$ the combined input capacity of its children, i.e. the sum of capacities of the at most two recursive calls to Divide-and-Conquer, is (by applying Lemma 4.6.6 for y_1^{DC} and Lemma 4.6.4 for the last inequality) at most

$$\begin{aligned}\tilde{c} - u_2 - y_1^B + \tilde{c} - y_1^B - y_1^{DC} - y_2^B \leq \\ \tilde{c} - u_2 - y_1^B + \tilde{c} - y_1^B - (\tilde{c} - y_1^B - u_2 - \varepsilon c) - y_2^B = \\ \tilde{c} - y_1^B - y_2^B + \varepsilon c \leq \tilde{c}.\end{aligned}$$

Performing the same argument iteratively for all nodes from the root downwards, this means that for all nodes with equal level the sum of their capacities remains bounded by c.

Now the argumentation is analogous to the proof of Theorem 3.3.1. For the sake of completeness we give an explicit argument. There are $m_\ell \leq 2^\ell$ nodes with level ℓ in the tree. Denoting the capacity of a node i in level ℓ by $\tilde{c}_\ell^i$ it was shown above that

$$\sum_{i=1}^{m_\ell} \tilde{c}_\ell^i \leq c.$$

Therefore, the total computation time for all nodes with level ℓ is bounded in complexity by

$$\sum_{i=1}^{m_\ell} \frac{\rho}{2^\ell} \cdot \frac{\bar{c}_\ell^i}{\varepsilon c} \leq \frac{\rho}{2^\ell} \cdot \frac{1}{\varepsilon}.$$

Summing up over all levels this finally yields

$$\sum_{\ell=0}^{\log \rho} \frac{\rho}{2^\ell} \cdot \frac{1}{\varepsilon} \leq 2\rho \cdot \frac{1}{\varepsilon}$$

which is of order $O(\min\{n \cdot 1/\varepsilon, 1/\varepsilon^2 \log(1/\varepsilon)\})$ and proves the theorem. □

4.7 Computational Results: FPTAS

In this section experimental results by Kellerer et al. [268] for the practical behavior of the fully polynomial approximation scheme FPSSP for (SSP) presented in Section 4.6 are presented.

In order to test the algorithm on instances in which it may exhibit a very bad performance three classes of data instances have been considered in [268].

- Problems p14: w_j uniformly random distributed in $]1, 10^{14}[$ and $c = 3 * 10^{14}$. Problems p14 are taken from Martello and Toth [332] where the range for the items was $(1, 10^5)$.
- Problems todd: $w_j = 2^{k+n+1} + 2^{k+j} + 1$ with $k = \lfloor \log_2 n \rfloor$, and $c = \lfloor \frac{1}{2} \sum_{j=1}^{n} w_j \rfloor$. Todd [86] constructed these problems such that any algorithm which uses LP-based upper bounding tests, dominance relations, and rudimentary divisibility arguments will have to enumerate an exponential number of states. Although the todd problems are difficult to solve for branch-and-bound methods, Ghosh and Chakravarti [173] showed that a local search algorithm based on a two-swap neighborhood always will find the optimal solution to these instances.
- Problems avis: $w_j = n(n+1) + j$, $c = \lfloor \frac{n-1}{2} \rfloor n(n+1) + \binom{n}{2}$. As noted in Section 4.4 Avis [86] showed that any recursive algorithm which does not use dominance will perform poorly for the avis problems.

Martello and Toth [335], report computational results both for the approximation schemes of Johnson [251] and Martello and Toth [331] and for the fully polynomial approximation schemes of Lawler [295] and Gens and Levner [167, 169]. Their results have been obtained on a CDC-Cyber 730 computer, having 48 bits available for integer operations. It is worth noticing that in their experiments the number of items

is at most 1000 and the error ε equals only $\frac{1}{2}, \frac{1}{4}$ and $\frac{1}{7}$. Hence, a direct comparison to the results of Kellerer et al. [268] is not possible.

The presented *FPTAS* was coded in FORTRAN 90 (Salford Version 2.18) and run on a INTEL PENTIUM, 200 MHZ with 32MB of RAM. Problems p14 have been tested for a number of items up to 5000 and a relative error ε equal to $\frac{1}{10}, \frac{1}{100}, \frac{1}{1000}$, respectively. For the same values of accuracy the todd problems have been tested for n equal to $10, 15, 20, 25, 30, 35$ (note the exploding size of the coefficients) and the avis problems for n up to 100000. While todd and avis are deterministic and thus only single instances are generated, for each accuracy and each number of items 10 instances have been generated for p14.

The results for p14, todd, and avis are discussed separately. The errors were determined by first computing the difference between the approximate solution and the upper bound c for p14 and for todd and avis with respect to the optimal solution value computed as in [335]. Then this difference was expressed as percentage of the respective upper bound. Naturally, the error is 0 for those problems where the optimal solution was found, while a running time equal to 0.000 means that less than 1/1000 seconds were required.

p14	$\varepsilon = 1/10$		$\varepsilon = 1/100$		$\varepsilon = 1/1000$	
n	per. error	time	per. error	time	per. error	time
10	1.422 (6.201)	0.320 (0.385)	0.164 (0.537)	0.091 (0.123)	0.043 (0.089)	4.151 (5.160)
50	0.4459 (1.524)	0.123 (0.384)	0.1207 (0.506)	0.692 (0.769)	0.0212 (0.0732)	28.479 (33.289)
100	0.232 (1.206)	0.059 (0.091)	0.1017 (0.343)	0.707 (0.767)	0.0152 (0.0478)	54.296 (60.492)
500	0.0368 (0.165)	0.056 (0.095)	0.0997 (0.336)	2.801 (3.131)	0.0331 (0.0879)	284.693 (325.317)
1000	0.023 (0.0977)	0.088 (0.126)	0.0231 (0.06)	3.900 (4.669)	0.0276 (0.0824)	551.059 (582.445)
5000	0.0063 (0.0158)	0.823 (0.843)	0.0065 (0.0153)	6.257 (7.526)	0.0074 (0.0203)	1927.779 (2010.602)

Table 4.4. Problems p14: Percentage errors (average (maximum) values) and computational times in seconds (average (maximum) values).

Table 4.4 shows the results for p14. The table gives four types of entries: In the first column the average and the maximum (in parentheses) percentage errors are reported; the second column shows the average and maximum (in parentheses) running times.

A detailed analysis of structure and depth of the binary tree generated by the Divide-and-Conquer recursions illustrates the contribution of this step for determining

the final solution. Tables 4.5 shows for each pair (n, ε) the minimum, average and maximum values of the maximum depth of the tree and the minimum, average and maximum number of times (the values given into brackets) procedure Divide-and-Conquer was called.

p14	$\varepsilon = 1/10$			$\varepsilon = 1/100$			$\varepsilon = 1/1000$		
n	min	average	max	min	average	max	min	average	max
10	0 (0)	0.4 (0.4)	1 (1)	0 (0)	0.7 (0.7)	1 (1)	0 (0)	0.2 (0.2)	1 (1)
50	0 (0)	1 (1)	2 (2)	1 (1)	1.4 (1.4)	3 (3)	1 (1)	1.2 (1.2)	2 (2)
100	1 (1)	1.1 (1.1)	2 (2)	1 (1)	1.9 (2)	3 (4)	0 (0)	1.8 (2.1)	2 (3)
500	0 (0)	0.8 (0.8)	1 (1)	2 (2)	3.2 (3.9)	4 (7)	2 (2)	3.2 (4.5)	4 (8)
1000	0 (0)	1.1 (1.1)	2 (2)	2 (2)	3.4 (4.1)	5 (7)	2 (2)	4.1 (6.6)	5 (14)
5000	0 (0)	0.9 (0.9)	1 (1)	3 (3)	4.5 (5.8)	6 (10)	3 (4)	5 (7.2)	6 (10)

Table 4.5. Problems p14: Maximum depth in the tree and number of calls to Divide-and-Conquer (in brackets), minimum, average and maximum values.

The number of calls to Divide-and-Conquer proportionally increases with the number of items involved and with the accuracy considered. In the instances of p14, Divide-and-Conquer was called a maximum number of 14 times for the case $n = 1000$ and $\varepsilon = 1/1000$. On the contrary, the maximum depth of the tree is never larger than 8 and seems to be only moderately dependent on the number of items.

Tables 4.6 and 4.7 refer to todd and avis. The entries in these tables give the percentage error and the computational time for each trial. Computational results show that the algorithm has an average performance much better than for these worst-case examples. Still compared to previous approximation approaches, the algorithm by Kellerer et al. [268] is also successful on these instances, requiring e.g. for problems todd never more than 20 seconds.

Finally, as shown in Table 4.7, for the avis problems, the algorithm generates decreasing errors when n increases. In particular, the instances become "easy", i.e. only algorithm Greedy is applied, when $\frac{1}{\varepsilon} \leq \frac{n-1}{2}$. In Table 4.7 all the instances with $\varepsilon = \frac{1}{10}$ and $n \geq 50$, $\varepsilon = \frac{1}{100}$ and $n \geq 500$, $\varepsilon = \frac{1}{1000}$ and $n \geq 5000$, respectively, only required Step 4 of algorithm FPSSP. For this reason the results for $n = 10000, 50000$ and 100000 are not shown.

todd	$\varepsilon = 1/10$		$\varepsilon = 1/100$		$\varepsilon = 1/1000$	
n	per. error	time	per. error	time	per. error	time
10	0.000	0.059	0.267	0.079	0.000	5.616
15	0.733	0.073	0.292	0.112	0.000	6.585
20	4.776	0.075	0.292	0.174	0.065	8.618
25	0.000	0.058	0.118	0.176	0.088	10.688
30	3.225	0.058	0.806	0.312	0.025	12.734
35	0.000	0.066	0.759	0.266	0.019	19.351

Table 4.6. Problems todd: Percentage errors and times in seconds. Single instances.

avis	$\varepsilon = 1/10$		$\varepsilon = 1/100$		$\varepsilon = 1/1000$	
n	per. error	time	per. error	time	per. error	time
10	2.953	0.033	0.000	0.106	0.000	5.296
50	1.004	0.092	0.519	0.751	0.014	45.154
100	0.501	0.179	0.490	0.733	0.000	73.532
500	0.100	0.872	0.100	0.877	0.050	404.215
1000	0.050	1.740	0.050	1.754	0.050	414.755
5000	0.010	0.009	0.010	0.067	0.010	0.644
10000	0.050	0.018	0.005	0.140	0.005	1.319

Table 4.7. Problems avis: Percentage errors and times in seconds. Single instances.

5. Exact Solution of the Knapsack Problem

Assume that a set of n items is given, each item j having an integer profit p_j and an integer weight w_j. The knapsack problem asks to choose a subset of the items such that their overall profit is maximized, while the overall weight does not exceed a given capacity c. Introducing binary variables x_j to indicate whether item j is included in the knapsack or not the model may be defined:

$$\text{(KP)} \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{5.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \tag{5.2}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n. \tag{5.3}$$

The assumptions on input data presented in Section 1.4 are assumed to hold, as, without loss of generality, we may use the transformation techniques described in that section to achieve this goal. This means that we may assume that every item j should fit into the knapsack, that the overall weight sum of the items exceeds c, and that all profits and weights are positive

$$w_j \leq c, \quad j = 1,\ldots,n, \tag{5.4}$$

$$\sum_{j=1}^{n} w_j > c, \tag{5.5}$$

$$p_j > 0, w_j > 0, \quad j = 1,\ldots,n. \tag{5.6}$$

Many industrial problems can be formulated as knapsack problems: Cargo loading, cutting stock, project selection, and budget control to mention a few examples. Several combinatorial problems can be reduced to (KP), and (KP) has important applications as a subproblem of several algorithms of integer linear programming. For a more thorough treatment of applications see Chapter 15.

It was pointed out in Section 1.5 that the knapsack problem belongs to the class of $\mathcal{NP}$-hard problems (see the Appendix A for a more extensive treatment). Moreover, there are even lower bounds available from theoretical computer science, which give

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

a theoretical indication that the running time of an exact algorithm for (KP) can not beat a certain threshold under reasonable assumptions on the model of computation. For a thorough treatment of this aspect of the theory of computation the interested reader is referred to the monograph by Blum et al. [39]. We restrict ourselves to pointing out some of the most important results in this direction.

Dobkin and Lipton [110] showed that under the linear decision tree model (LDT), sometimes also referred to as linear search algorithms (LSA), the running time of every exact algorithm for (KP) (in fact even for the decision version of (SSP)) is bounded from below by an expression proportional to n^2. In the famous paper by Ben-Or [33] this lower bound was extended to the algebraic computation tree model (ACT). The same n^2 bound is also valid for the *random access machine* model (RAM) as shown by Klein and Meyer auf der Heide [275]. An alternative result for a computation model with real numbers was given by Brimkov and Dantchev [55]. Their lower bound is proportional to $f(w_1,\ldots,w_n)n\log n$ for an arbitrary continuous function f.

Although $\mathcal{NP}$-hard, (KP) belongs to the "more pleasant" problems of this class named $\mathcal{NP}$-hard in the weak sense, as they may be solved in pseudopolynomial time. It would be a welcome justification for the construction of these not strictly polynomial algorithms to have lower bounds showing that at least under a weak model of computation no polynomial algorithms can exist. However, it could be shown by Meyer auf der Heide [346] that for the linear decision tree model no super-polynomial lower bound can exist. This in some sense negative result was extended by Fournier and Koiran [149] who showed that for even less powerful models of computation no super-polynomial lower bound is likely to exist.

From practical experience it is known that many (KP) instances of considerable size can be solved within reasonable time by exact solution methods. This fact is due to several algorithmic refinements which emerged during the last two decades. This includes *primal-dual dynamic programming recursions*, the concept of solving a *core*, and the separation of *cover inequalities* to tighten the formulation. For a recent overview of the latest techniques see Martello, Pisinger and Toth [322].

For problem instances with a large number of items and weights which are relatively small compared to the knapsack capacity, it can even be expected that the simple algorithm Greedy as introduced in Section 2.1 yields a solution value not far away from the optimal value.

From a theoretical point of view, the related problem of finding necessary and sufficient conditions on the set of weights $\{w_1,\ldots,w_n\}$, such that Greedy delivers an optimal solution *for all* given profit values, was considered by several authors for different variants of knapsack problems. In particular the contributions by Magazine, Nemhauser and Trotter [313], Hu and Lenhard [239], Tien and Hu [458] and Vizvari [475] should be mentioned.

A more formal approach to this problem was pursued by Cerdeira and Barcia [74] who settled the question for (KP) by presenting a precise characterization of the

weight sets where the set of feasible solutions form a matroid. By definition, these are exactly the instances where Greedy delivers the optimal solution value for arbitrary profit values. Naturally, the corresponding conditions on the weights are fairly restrictive.

It should be noted that these conditions can be verified in polynomial time. Cerdeira and Barcia [74] showed that deciding for a given instance of (KP) (and even (SSP)) in polynomial time whether Greedy yields an optimal solution or not would have the very unlikely consequence of $\mathcal{P}=\mathcal{NP}$.

Recall that the two classical optimal solution methods for (KP), namely dynamic programming and branch-and-bound, were introduced in Sections 2.3 and 2.4. In Chapter 6 we will go into more details with approximation algorithms, and present the best currently known *PTAS* and *FPTAS*.

5.1 Branch-and-Bound

Recalling the main principles of branch-and-bound algorithms from Section 2.4 we will use the term *fixed variables* to denote those variables which have been assigned a value during the branching process, while we will use the term *free variables* to denote the remaining variables.

It is quite obvious from the description of the branch-and-bound paradigm that the main ingredients for a successful branch-and-bound algorithm are "good" upper bounds, i.e. functions which for a given subset X of the solution space derives a valid upper bound U as close to z^* as possible.

We will briefly review some of the bounding methods given in the literature. Most of them are based on the relaxations introduced in Section 3.8. Note that some bounds were used independently by several authors which impedes the exact assignment of the original references.

5.1.1 Upper Bounds for (KP)

Most of the upper bounds for (KP) rely on some kind of sorting according to the efficiency e_j of an item j defined as $e_j := \frac{p_j}{w_j}$. For simplicity of the presentation we will in the following assume that the items are sorted according to decreasing efficiencies

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}, \tag{5.7}$$

although most of the bounds can be derived faster without this sorting.

The simplest upper bound for (KP) can be derived by relaxing the constraint $x_1 \in \{0,1\}$ on the most efficient item to the weaker constraint $x_1 \geq 0$. This leads to the trivial bound

$$U_0 := \left\lfloor \frac{cp_1}{w_1} \right\rfloor . \tag{5.8}$$

Since all profits are assumed to be integers any optimal solution value of (KP) will be integral, and thus we round down the bound U_0 as stated. If the items are already sorted according to decreasing efficiencies (5.7), U_0 may be derived in constant time, otherwise the most efficient item must first be determined in $O(n)$ time. For small capacities c the bound U_0 is reasonably close to z^*, and the fast evaluation makes it applicable inside branch-and-bound algorithms: Assuming that several variables already have been fixed to a value due to the branching, the bound U_0 is derived on the set of free variables in constant time. The natural generalization of U_0 for a *over-filled knapsack* is to remove an appropriate fraction of the least efficient item j.

The LP-relaxation (LKP) of (KP) as introduced in Section 2.2 leads to the so-called *Dantzig bound* [97], which may be derived by use of Theorem 2.2.1. As in (5.8), we may round down the objective of the LP-relaxation thus getting the bound

$$U_1 := U_{\mathrm{LP}} := \lfloor z^{LP} \rfloor . \tag{5.9}$$

We observe that $U_1 \leq U_0$. The efficient computation of the split item and hence of z^{LP} in $O(n)$ time was discussed in Section 3.1. Inside a branch-and-bound algorithm, the items will normally be sorted according to (5.7) and hence U_1 may be derived in a straightforward way in $O(n)$ time. If we round down the obtained LP-solution vector to an integer solution we get the *split solution* $\hat{x}$ which has the profit and weight sums

$$\hat{p} = \sum_{j=1}^{n} p_j \hat{x}_j, \qquad \hat{w} = \sum_{j=1}^{n} w_j \hat{x}_j, \tag{5.10}$$

as also defined in Section 2.2.

A different bound can be obtained through Lagrangian Relaxation. For the knapsack problem we can only relax the single capacity constraint with a nonnegative multiplier λ yielding $L(\mathrm{KP},\lambda)$ defined by

$$\begin{aligned} L(\mathrm{KP},\lambda) \qquad & \text{maximize} \sum_{j=1}^{n} p_j x_j + \lambda \left(c - \sum_{j=1}^{n} w_j x_j \right) \\ & \text{subject to } x_j \in \{0,1\}, \qquad j = 1,\ldots,n. \end{aligned} \tag{5.11}$$

As the remaining linear constraints in $L(\mathrm{KP},\lambda)$ correspond to the convex hull of the set of integer solutions to $L(\mathrm{KP},\lambda)$, the Lagrangian Relaxation, however, cannot return a tighter bound than the LP-relaxation [489, Theorem 10.3]. Indeed a bound equal to that of the LP-relaxation may be obtained by choosing $\lambda = \frac{p_s}{w_s}$. Hence the best bound obtained by $L(\mathrm{KP},\lambda)$ equals U_1.

Considering separately the case of packing the split item s or not, Martello and Toth [324] derived the following upper bound

$$U_2 := \max\left\{\left\lfloor \hat{p} + \left(c - \hat{w}\right)\frac{p_{s+1}}{w_{s+1}}\right\rfloor, \left\lfloor \hat{p} + p_s + \left(c - \hat{w} - w_s\right)\frac{p_{s-1}}{w_{s-1}}\right\rfloor\right\}, \quad (5.12)$$

where $\hat{p}$ and $\hat{w}$ are given by (5.10). The first term in (5.12) appears by imposing the additional constraint $x_s = 0$ and deriving bound U_0 on the remaining items $j \geq s+1$. The second term appear in a similar way by imposing $x_s = 1$ and deriving bound U_0 on items $j \leq s-1$. As the inclusion of item s means that the knapsack gets overfilled we remove a fraction of the least efficient item $j \leq s-1$. It is easy to see that U_2 can be computed in $O(n)$ time as we only need to know $s-1, s$ and $s+1$. Moreover, we observe that $U_2 \leq U_1$.

Generalizing this principle, Martello and Toth [333] proposed the derivation of arbitrarily tight upper bounds by using *partial enumeration*. Assume that the items are sorted according to decreasing efficiencies (5.7), let $M \subseteq \{1,\ldots,n\}$ be a subset of the items and assume that $X_M = \{x_j \in \{0,1\},\ j \in M\}$. Then every optimal solution of (KP) must have its origin in one of the zero-one vectors in X_M and an upper bound on (KP) is thus given by

$$U_M := \max_{\tilde{x} \in X_M} U(\tilde{x}), \quad (5.13)$$

where $U(\tilde{x})$ is any upper bound on (KP) with the additional constraint $x_j = \tilde{x}_j$ for $j \in M$. Since the computational effort to compute U_M is $O(2^{|M|})$ times the complexity of deriving $U(\tilde{x})$, this approach is only useful for small sets M. Indeed, for $M = \{1,\ldots,n\}$ we would compute the optimal solution value z^* but the solution time would be huge. A good value of this bound can be achieved by choosing as M a small subset of items with efficiency close to that of the split item s. (cf. the core concept in Section 5.4).

Other more or less straightforward bounds along similar or related lines were derived by Fayard and Plateau [138], Müller-Merbach [354] and Dudzinski and Walukiewicz [117]. Basically all the above bounds including the Martello and Toth upper bound appear trivially by running a branch-and-bound algorithm to a fixed depth of the search tree, choosing the smallest upper bound of the leaf nodes. The only difference between the bounds is the depth of the search tree and the bounds used at the leaf nodes. None of the bounds are very well suited for use in a branch-and-bound algorithm, as the time used for deriving the more complex bounds corresponds to that of exploiting the same search tree with a simpler bound.

A completely different approach to derive upper bounds for (KP) appears by strengthening the LP-relaxation (LKP) of (KP). In this approach we add valid inequalities to (LKP) which are redundant to the IP-formulation but which "cut off" the LP-optimal solution x^{LP}. (See Section 15.3 for more details). In this way they lead to a new and smaller optimal LP-solution value z^{LP1}. The study of facets of the knapsack polytope dates back to Balas [18], Hammer, Johnson and Peled [207]

and Wolsey [488] who gave necessary and sufficient conditions for a canonical inequality to be a facet of the knapsack polytope. Balas and Zemel [22] made the first experiments adding facets to the formulation of so-called *strongly correlated knapsack problems*, but at that time the solution times were too large to show the benefits of this approach. For an introduction to the theory of polyhedral properties of the knapsack problem see Section 3.10.

Rather natural constraints which may be added to (LKP) are *cardinality constraints* [18]. These consist of an upper and lower bound on the number of items in the optimal solution. It is easy to see that the maximal number of items in a feasible packing is given be selecting the items with smallest weight until they exceed the knapsack capacity. Hence, after sorting the items in increasing order of item weights, we can define k as

$$k := \min\left\{h \middle| \sum_{j=1}^{h} w_j > c\right\} - 1, \tag{5.14}$$

and add a *maximum cardinality constraint* in form of the equation

$$\sum_{j=1}^{n} x_j \leq k \tag{5.15}$$

to (KP) without excluding any feasible integer solution. Note, that this addition will only improve the bound resulting from the (LKP) if constraint (5.15) is violated by the optimal solution of (LKP). This means that it should only be imposed if $k \leq s-1$.

In a similar way we may define *minimum cardinality constraints* as presented by Martello and Toth [336]. Assume that some lower bound z^ℓ is given, e.g., through heuristic approaches, and that the items are ordered according to decreasing profit values. Setting

$$k := \max\left\{h \middle| \sum_{j=1}^{h} p_j \leq z^\ell\right\} + 1, \tag{5.16}$$

we can add the constraint

$$\sum_{j=1}^{n} x_j \geq k \tag{5.17}$$

to (KP) which is fulfilled by all feasible solutions with objective value larger than z^ℓ. Again, adding (5.17) makes sense only if it is violated by the optimal solution of (LKP) which is the case for $k \geq s$.

In the following we will only consider maximum cardinality constraints as the treatment of minimum cardinality constraints is symmetric. Adding constraint (5.15) to (LKP) one gets the *cardinality constrained knapsack problem* (kKP) which is studied in more detail in Section 9.7.

$$\begin{aligned} (\text{C}k\text{KP}) \qquad \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & \sum_{j=1}^{n} x_j \leq k, \\ & 0 \leq x_j \leq 1, \quad j = 1,\ldots,n. \end{aligned} \tag{5.18}$$

The LP-solution of this problem leads to a bound U_3 where $U_3 \leq U_1$. As the bound needs to be derived several times during a branch-and-bound algorithm, Martello and Toth [336] chose to further relax this problem in order to get a quickly derivable (but weaker) bound. Lagrangian relaxing the cardinality constraint using a nonnegative multiplier $\lambda \geq 0$, one gets the problem $L(\text{C}k\text{KP},\lambda)$ given by

$$\begin{aligned} L(\text{C}k\text{KP},\lambda) \qquad \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j - \lambda \left(\sum_{j=1}^{n} x_j - k \right) = \sum_{j=1}^{n} \tilde{p}_j x_j + \lambda k \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & 0 \leq x_j \leq 1, \qquad j = 1,\ldots,n, \end{aligned} \tag{5.19}$$

which is an (LKP) with (possibly negative) item profits $\tilde{p}_j = p_j - \lambda$. The problem may easily be solved in $O(n)$ using the transformation from Section 1.4 followed by Lemma 3.1.1. We are interested in deriving the value of λ for which the tightest (i.e. smallest) bound is obtained through $L(\text{C}k\text{KP},\lambda)$. Thus we solve the *Lagrangian dual problem*

$$U_4 = \min_{\lambda \geq 0} L(\text{C}k\text{KP},\lambda). \tag{5.20}$$

Obviously $U_4 \leq U_1$ as it contains U_1 as a special case for $\lambda = 0$.

It is well-known that the Lagrangian dual as a function of λ is piecewise linear and convex [489, Section 10.2]. The break-points (non-differentiable points) correspond to the situations where a new item becomes the split item when solving (5.19). Two items i and j may interchange the role of being the split item when their efficiency is the same i.e. when $\frac{p_i-\lambda}{w_i} = \frac{p_j-\lambda}{w_j}$. As there are $O(n^2)$ values of λ for which this may happen, we may list all these candidate values, sort them, and use binary search in the set to find the value of λ which minimizes $L(\text{C}k\text{KP},\lambda)$. This can be done in $O(n^2)$ time, as we in each iteration need to solve the problem (5.19) in $O(n)$ time, and the binary search needs $O(\log n)$ iterations. The most expensive part is to find the median of the candidate set of the λ-values. This takes $O(n^2)$ time in the first iteration, and the effort is halved in each succeeding iteration.

One may also notice that the maximum value of λ corresponds to the k-th largest profit p_j in (5.19) and the minimum value of λ is 0. Thus, to solve the Lagrangian

dual we may use binary search in the linear interval $[0, p_{\max}]$. This will demand $O(nb)$ time, where b is number of bits used to represent λ. If λ is an integer, one will use at most $O(n \log p_{\max})$ time.

In [336] Martello and Toth also discuss a third algorithm, which despite the worst-case complexity of $O(n^3)$ is claimed to have excellent performance in practice.

A different approach used by Martello, Pisinger and Toth [321] is to surrogate relax (kKP). Using the multipliers 1 and $\mu \geq 0$ for the two constraints one gets the problem

$$\begin{aligned} S(Ck\text{KP},\mu) \qquad & \text{maximize} \sum_{j=1}^{n} p_j x_j \\ & \text{subject to} \sum_{j=1}^{n} (w_j + \mu) x_j \leq c + k\mu, \\ & 0 \leq x_j \leq 1, \qquad j = 1,\ldots,n, \end{aligned} \tag{5.21}$$

which is also an (LKP), solvable in $O(n)$ by use of Lemma 3.1.1. The problem of finding the value of μ for which the tightest bound is obtained, is symmetric to the Lagrangian case. Hence, we solve the *surrogate dual problem*

$$U_5 = \min_{\mu \geq 0} S(Ck\text{KP},\mu) \tag{5.22}$$

The profits and weights are positive for any value of $\mu \geq 0$, hence the surrogate variant is slightly easier to solve. As in the Lagrangian relaxation case described above, we notice that the only interesting values of μ are those where two items i, j interchange the role of being the split item when solving the LP-relaxation. This may occur when items i and j have the same efficiency, i.e. when $\frac{p_i}{w_i+\mu} = \frac{p_j}{w_j+\mu}$, so there are again $O(n^2)$ values of μ to be investigated. The maximum value of μ is bounded by $p_{\max} w_{\max} - 1$ corresponding to the case where the most efficient item $(p_{\max}, 1)$ and least efficient item $(1, w_{\max})$ interchange position when solving (5.21). Martello, Pisinger and Toth notice that for most purposes it is sufficient to determine μ to integer precision. In this way one may use binary search over integer values of μ in the interval $[0, p_{\max} w_{\max} - 1]$ which can be done in $O(n \log p_{\max} + n \log w_{\max})$ time.

5.1.2 Lower Bounds for (KP)

In order to solve the (KP) through branch-and-bound, a lower bound z^ℓ is needed which the upper bounds can be tested against. Although one may initially set $z^\ell = 0$ good branch-and-bound implementations rely on being able to construct a lower bound close to the optimal solution. The simplest lower bound is the split solution which appears from the LP solution by truncating x_s to zero (see algorithm Greedy-Split, Section 2.1). The objective value of the split solution $\hat{x}$ is

$$z' = \hat{p} \tag{5.23}$$

Fixed-cardinality heuristics have been proposed in Pisinger [381]. These heuristics are particularly suited for so-called *strongly correlated knapsack problems*. The *forward greedy solution* is the best value of the objective function when adding one item to the split solution:

$$z^f := \max_{j=s+1,\ldots,n} \{\hat{p} + p_j \,|\hat{w} + w_j \leq c\}, \tag{5.24}$$

and the *backward greedy solution* is the best value of the objective function when adding the split item to the split solution and removing another item:

$$z^b := \max_{j=1,\ldots,s-1} \{\hat{p} + p_s - p_j \,|\hat{w} + w_s - w_j \leq c\}. \tag{5.25}$$

The time complexity of bounds z^f, z^b is $O(n)$ but the performance ratio is arbitrarily close to zero.

Several of the approximation techniques presented in Section 2.5 and Chapter 6 can also be used to derive a lower bound z^ℓ for (KP), however at the cost of a higher computational cost.

5.1.3 Variable Reduction

Before running a branch-and-bound algorithm or dynamic programming algorithm it may be advantageous to fix some variables at their optimal values. In this way the number of items n may be reduced, and hence the expensive enumeration will take less time. Since the worst-case time complexity of branch-and-bound and dynamic programming is very bad (exponential time, resp. pseudopolynomial time), any effort possible should be made to reduce the instance size.

The basic principle of reduction was described in Section 3.2. For each binary variable in (KP) we test the two possible assignments of the variable, deriving an upper bound U for each of the branches. If one of the branches cannot lead to an improved solution (i.e. $U \leq z^\ell$) we may fix the variable to the opposite value. Ingargiola and Korsh [245] presented the first reduction algorithm for the (KP), but several improvements have emerged during the years.

For the following discussion assume that a lower bound z^ℓ has been found through any kind of heuristic and that the corresponding solution vector x has been saved. The reason for saving the optimal solution vector is that it allows us to use tighter reductions, since we are only interested in solutions which may *improve* the lower bound.

Let U_j^0 be an upper bound for (KP) with additional constraint $x_j = 0$. Similarly let U_j^1 be an upper bound for (KP) with constraint $x_j = 1$. Then the set of items which can be fixed at optimal value $x_j = 1$ is

$$N^1 := \{j \in N : U_j^0 < z^\ell + 1\}. \tag{5.26}$$

In a similar way the set of items which can be fixed at $x_j = 0$ is

$$N^0 := \{j \in N : U_j^1 < z^\ell + 1\}. \tag{5.27}$$

If $N^1 \cup N^0 = \emptyset$ or if $\sum_{j \in N^1} w_j > c$ then z^ℓ is optimal and we may terminate.

The remaining items $F = \{1,\ldots,n\} \setminus (N^1 \cup N^0)$ are called *free*, and a reduced problem appears for $F \neq \emptyset$ as follows, by setting $\tilde{p} := \sum_{j \in N^1} p_j$, and $\tilde{w} := \sum_{j \in N^1} w_j$:

$$\begin{aligned} \text{(KPR)} \qquad & \text{maximize} \sum_{j \in F} x_j + \tilde{p} \\ & \text{subject to} \sum_{j \in F} w_j x_j \leq c - \tilde{w}, \\ & x_j \in \{0,1\}, \qquad j \in F. \end{aligned} \tag{5.28}$$

The optimal solution value is found as $z^* = \max\{z^\ell, z(\text{KPR})\}$, where z^ℓ is the lower bound used in the reduction. Notice, that for most bounds there is no reason for deriving U_j^1 for $j < s$ since the additional constraint $x_j = 1$ will not affect the bound, and hence it will not be possible to reduce any variables. Similarly, deriving U_j^0 for $j > s$ is obsolete for bounds appearing by linear relaxation.

The time complexity of the reduction depends on the complexity of the applied bound. Dembo and Hammer [104] used the following bounds which can be derived in constant time once the split item s has been found

$$U_j^0 := \hat{p} - p_j + \left(c - \hat{w} + w_j\right)\frac{p_s}{w_s} \qquad U_j^1 := \hat{p} + p_j + \left(c - \hat{w} - w_j\right)\frac{p_s}{w_s}. \tag{5.29}$$

Both bounds appear by relaxing the constraint $x_s \in \{0,1\}$ such that x_s becomes an unbounded (positive or negative) real variable. Using these bounds the whole reduction runs in $O(n)$ time.

Ingargiola and Korsh [245] proposed a reduction with tighter but more time-consuming bounds, by defining U_j^0 and U_j^1 as the LP-bound for (KP) with additional constraint $x_j = 0$, resp. $x_j = 1$. Since U_1 can be derived in $O(n)$ time for each item j, the total running time becomes $O(n^2)$ for the reduction. Notice that the split item s_j^0 (resp. s_j^1) may be different for each item j considered.

The running time of the Ingargiola and Korsh reduction may be improved by sorting the items according to decreasing efficiencies before the reduction. Instead of deriving the Dantzig bound by searching for the split item s as given in (2.5) one simply starts from the present split item s and moves forward or backward depending on the new capacity of the knapsack, updating the profit and weight sums as items are passed. For most instances appearing in practice this will mean that only a few steps forward or backward are needed, resulting in running times close to linear for the whole reduction. The worst-case complexity is unchanged $O(n^2)$.

Martello and Toth [333] used the bound U_2 to derive the bounds U_j^0 and U_j^1. In order to improve the worst-case time complexity of deriving the bounds, they sorted the items according to decreasing efficiencies and derived the following sums $\overline{p}_j = \sum_{i=1}^{j} p_i$ resp. $\overline{w}_j = \sum_{i=1}^{j} w_i$ for each item j. In this way binary search could be used for finding the new split item s' in time $O(\log n)$. This means that the worst-case running time of the whole reduction is $O(n \log n)$. Note, that the same approach may be used to improve the worst-case complexity of the Ingargiola and Korsh reduction algorithm.

Finally, Pisinger [383] used the enumerative bound U_M given by (5.13) for deriving the bounds U_j^0 and U_j^1. Although this bound is computationally very expensive, it could be derived in pseudopolynomial time $O(c)$ as part of the primal-dual dynamic programming algorithm described in Section 5.3. The reduction was only applied on variables j where simpler and more quickly derivable bounds had failed to reduce the variable.

As the effect of the reduction strictly depends on the quality of the lower bound z^ℓ, Ingargiola and Korsh proposed to improve z^ℓ during the reduction. Since each split item corresponds to a greedy solution, we may set $z^\ell := \max\{z^\ell, z^0, z^1\}$ with

$$\begin{aligned} z^0 &:= \max_{j=1,\ldots,s-1} \left(-p_j + \sum_{i=1}^{s_j^0-1} p_i \right), \\ z^1 &:= \max_{j=s+1,\ldots,n} \left(p_j + \sum_{i=1}^{s_j^1-1} p_i \right). \end{aligned} \tag{5.30}$$

Following this direction, Martello and Toth [333] proposed to first find and save the split items s_j^0 and s_j^1 for all additionally constrained problems, and during this process improve the lower bound z^ℓ. Only when this lower bound z^ℓ has been derived, the reduction is performed.

5.1.4 Branch-and-Bound Implementations

To our knowledge, the first branch-and-bound algorithm for (KP) was presented in 1967 by Kolesar [283]. Several variants of the basic framework have emerged since then, usually based on a *depth-first search* to limit the number of open nodes to $O(n)$ at any stage of the enumeration. Moreover, a greedy strategy where the most efficient of the free variables is considered in each branching node, seems to lead to best solution times. All algorithms assume that the items are sorted according to decreasing efficiencies (5.7) as the sorting time $O(n \log n)$ for small instances is negligible compared to the branching effort. For large-sized instances it may be sufficient to sort only a small subset of the items as will be described in Section 5.4.

Among the branch-and-bound algorithms we may distinguish between two variants: A *primal* method which adds items to an initially empty knapsack until the weight constraint is exceeded, and a *primal-dual* approach which accepts infeasible solutions in a transition stage. In principle one could also have a *dual* method where all items initially were placed in the knapsack and items were repeatedly removed in the branch-and-bound process.

Most primal methods are based on the framework proposed by Horowitz and Sahni [238] which is illustrated in Figure 5.1. In a recursive formulation each iteration of the Primal-Branch algorithm corresponds to a dichotomic branch on the most efficient free variable x_b, where $x_b = 1$ is investigated before $x_b = 0$ according to the greedy principle. Having branched on variables x_j, $j < b$, the profit and weight sum of the currently fixed variables is $\overline{p}$ resp. $\overline{w}$, hence if $\overline{w} > c$ for a given node, we may backtrack. Also, if an upper bound for the remaining problem is less or equal to the current lower bound z^ℓ, we may backtrack. The lower bound z^ℓ is improved during the search and the algorithm returns true if an improved solution was found in the present node or any descendant node. This makes it possible to define the optimal solution vector x^* during the backtracking as will be described below. At the beginning the lower bound should be initialized to $z^\ell := 0$ before calling Primal-Branch$(1,0,0)$.

```
Algorithm Primal-Branch(b, p̄, w̄): boolean
improved := false
if w̄ > c then return improved
if p̄ > z^ℓ then z^ℓ := p̄, b* := b, improved := true
if j > n or c − w̄ < min_{i=j,...,n} w_i then return improved
derive upper bound U_0 with capacity c − w̄
if p̄ + U_0 ≤ z^ℓ then return improved
if Primal-Branch(b + 1, p̄ + p_b, w̄ + w_b) then x_b := 1, improved := true
if Primal-Branch(b + 1, p̄, w̄) then x_b := 0, improved := true
return improved
```

Fig. 5.1. The Primal-Branch algorithm is a recursive branch-and-bound algorithm which in each step branches on the variable x_b. The local variable improved is used to indicate whether an improved solution was found at the present or descending branching nodes. The Primal-Branch algorithm returns the value of improved.

The algorithm backtracks if we have reached the bottom of the search tree ($b > n$), or if the bound U_0 shows that no improved solution can be found in descending nodes ($\overline{p} + U_0 \leq z^\ell$). The bound U_0 is derived on the free variables, hence x_b is used as the most efficient item in (5.8). We may also backtrack if no more item fit into the residual capacity of the knapsack, i.e. if $c - \overline{w} < \min_{j=b,\ldots,n} w_j$. The minimum weights $\min_{j=b,\ldots,n} w_j$ for all $b = 1,\ldots,n$ can be determined a-priori in $O(n)$ time and saved in a table, such that all the above test can be performed in constant time.

The optimal solution vector x^* is partially defined during backtracking of the algorithm. If an improved lower bound has been found at a node, Primal-Branch will return `true` in all parent nodes, thus forcing the solution vector x to be updated. In this way all nodes can be handled in constant time resulting in a running time of $O(2^n)$. If we instead saved the whole solution vector each time the lower bound was modified, the treatment of one node in the search tree would take $O(n)$ resulting in an overall running time of $O(n2^n)$.

We use the variable b^* to remember the index of the item considered, when the optimal solution was found. To define the whole optimal solution vector x^* we simply set $x_j^* = x_j$ for $j = 1,\ldots,b^*-1$ while $x_j^* = 0$ for $j = b^*,\ldots,n$.

The MT1 algorithm by Martello and Toth [324] is based on the same framework as Primal-Branch, using the U_2 upper bound instead of U_0 and a new dominance criterion to skip nodes which will not improve the solution. A comparison of several algorithms built within the above framework is presented in Martello and Toth [335] showing that the MT1 algorithm in general has the smallest running times. For most problem instances there are, however, only minor differences in the running times.

The so-called *primal-dual algorithm* was developed by Pisinger [381]. Starting from the split item s the Primal-Dual-Branch algorithm repeatedly inserts an item into the solution if the current weight sum is less than c. Otherwise it removes an item. In this way one only needs to consider knapsacks which are "appropriately filled". On the other hand infeasible solutions must be considered in a transition stage. This concept is closely related to the idea of balancing as described in Section 3.6. An outline of Primal-Dual-Branchis found in Figure 5.2.

The variables $x_{a+1},\ldots,x_{b-1}$ are fixed at some value and for the corresponding profit and weight sums $\overline{p}$ and $\overline{w}$, we implicitly assume that all items $j \le a$ were also packed into the knapsack. Thus we insert an item when $\overline{w} \le c$ and remove an item when $\overline{w} > c$. In contrary to the Primal-Branch algorithm we cannot backtrack immediately whenever $\overline{w} > c$, as infeasible solutions may become feasible at a later stage by removal of some items. Instead we must rely on the upper bound U_0 to prune the search tree. The resulting algorithm is called by Primal-Dual-Branch$(s-1,s,\hat{p},\hat{w})$, where $\hat{p}$ and $\hat{w}$ are the profits of the split solution defined in (5.10). Initially the lower bound must be set to $z^\ell := 0$.

When inserting item b the bound U_0 is derived on items $j \ge b$. When removing item a the knapsack is over-filled and thus U_0 is derived as described for this case in (5.8).

As in Primal-Branch, the Primal-Dual-Branch algorithm returns true if an improved solution has been found at the present node or any descendant node, thus forcing the optimal solution vector to be defined during backtracking. After completion of the algorithm, we set $x_j^* = x_j$ for $j = a^*,\ldots,b^*$ while $x_j^* = 1$ for $j = 1,\ldots,a^*-1$ and $x_j^* = 0$ for $j = b^*+1,\ldots,n$. This defines the optimal solution vector x^*.

```
Algorithm Primal-Dual-Branch(a, b, p̄, w̄): boolean
improved := false
if w̄ ≤ c then
    try to insert item b
    if p̄ > z^ℓ then z^ℓ := p̄, a* := a, b* := b, improved := true
    if b > n then return improved
    derive upper bound U0 with capacity c − w̄
    if p̄ + U0 ≤ z^ℓ then return improved
    if Primal-Dual-Branch(a, b+1, p̄ + p_b, w̄ + w_b) then
        x_b := 1, improved := true
    if Primal-Dual-Branch(a, b+1, p̄, w̄) then
        x_b := 0, improved := true
else
    try to remove item a
    if a < 1 then return improved
    derive upper bound U0 with capacity c − w̄
    if p̄ + U0 ≤ z^ℓ then return improved
    if Primal-Dual-Branch(a − 1, b, p̄ − p_a, w̄ − w_a) then
        x_a := 0, improved := true
    if Primal-Dual-Branch(a − 1, b, p̄, w̄) then
        x_a := 1, improved := true
return improved
```

Fig. 5.2. The primal-dual algorithm Primal-Dual-Branch starts from the split solution, and in each iteration it either inserts an item b or removes an item a according to the weight sum $\overline{w}$ of the currently included items. The local variable improved is used to indicate whether an improved solution was found at the present or descending branching nodes. The value of improved is returned from the algorithm.

Every node of Primal-Dual-Branch can be handled in constant time, and hence the complete solution time is of magnitude $O(2^n)$. The solution space of Primal-Dual-Branch will only consider nodes where $\overline{w}$ is between $c - w_{\max}$ and $c + w_{\max}$. This should be compared to the Primal-Branch algorithm where $\overline{w}$ could be between 0 and c. If the weights are of moderate size compared to the capacity c, algorithm Primal-Dual-Branch should search a smaller solution space.

The algorithm Primal-Dual-Branch lends itself well to the bound U_0. As the weight sum $\overline{w}$ will always be close to the capacity, U_0 will be quite close to the bound U_1 and hence we get a good bound in constant time.

5.2 Primal Dynamic Programming Algorithms

The primal dynamic programming algorithms presented in this section have the nice property, that they solve the (KP) for *all capacities*, and hence can be used for the applications mentioned in Section 1.3.

	$z_j(d)$	d 0	1	2	3	4
	0	0	0	0	0	0
j	1	0	0	0	1	1
	2	0	3	3	3	4

	states	$(\overline{w},\overline{p})$
	0	(0,0)
j	1	(0,0) (3,1)
	2	(0,0) (1,3) (4,4)

Fig. 5.3. An illustration of the Bellman recursion for the instance $\{(p_1,w_1),(p_2,w_2)\} = \{(1,3),(3,1)\}$ and $c = 4$. The left table shows the outcome of $z_j(d)$ using the Bellman recursion. The right table shows the corresponding lists of undominated states. Thus for e.g. $j = 2$ we have the list of undominated states $\{(0,0),(1,3),(4,4)\}$ where state $(1,3)$ means that a profit sum of 3 can be obtained with a capacity of 1.

The straightforward *Bellman recursion* [32] was introduced in Section 2.3. Let $KP_j(d)$ for $j = 0,\ldots,n$ and $d = 0,\ldots,c$ be the (KP) subproblem defined on items $1,\ldots,j$ and restricted capacity d as defined in (2.7). Moreover, let $z_j(d)$ be the optimal solution value corresponding to $KP_j(d)$. The Bellman recursion sets $z_0(d) = 0$ for $d = 0,\ldots,c$, using the following recursion to obtain subsequent values of $z_j(d)$

$$z_j(d) := \begin{cases} z_{j-1}(d) & \text{if } d < w_j, \\ \max\{z_{j-1}(d), z_{j-1}(d-w_j) + p_j\} & \text{if } d \geq w_j, \end{cases} \tag{5.31}$$

for $j = 1,\ldots,n$ and $d = 0,\ldots,c$ (see also (2.8)). The optimal solution value to (KP) is found as $z^* = z_n(c)$. The time and space complexity of the Bellman recursion in its direct form is $O(nc)$.

Using the concept of *states* as described in Section 3.4 computational effort may be decreased significantly for many instances, although the worst-case time complexity is unchanged. A state is a pair of integers $(\overline{w},\overline{p})$ where $\overline{w}$ denotes a given capacity, and $\overline{p}$ denotes the corresponding maximum profit sum obtainable $z_j(\overline{w})$. See Figure 5.3 for an example.

Notice that the Bellman recursion solves the *all-capacities knapsack problem*, since table $z_n(d)$ will hold the optimal solution value for each $d = 0,\ldots,c$.

5.2.1 Word RAM Algorithm

Pisinger [392] presented an improved variant of recursion (5.31) which runs in time $O(nm/\log m)$ on a word RAM, where $m = \max\{c,z^*\}$. The algorithm solves the all-capacities variant of the knapsack problem as defined in Section 1.3.

Recalling the basic principles of word RAM algorithms from Section 3.7 we will assume that the word size W is sufficiently large to hold the largest coefficients of the instance, i.e.

$$W \text{ in } \Theta(\log m) \tag{5.32}$$

The algorithm makes use of the same binary coding of the optimal solutions to (KP_j) as used for the subset sum version of the problem in Section 4.1.1, but having

$g(j,\overline{w})$		0	1	2	3	4
	0	1	0	0	0	0
j	1	1	0	0	1	0
	2	1	1	0	1	1

$h(j,\overline{p})$		0	1	2	3	4
	0	1	0	0	0	0
j	1	1	1	0	0	0
	2	1	1	0	1	1

$g(j,\overline{w})$		0	1	2	3	4
	0	1	0	0	0	0
j	1	1	0	0	1	0
	2	1	1	0	0	1

$h(j,\overline{p})$		0	1	2	3	4
	0	1	0	0	0	0
j	1	1	1	0	0	0
	2	1	0	0	1	1

Fig. 5.4. Using the same instance as in Figure 5.3 we get the above tables. The two tables at the top show the (wrong) outcome if the recursion is used on the weights and profits independently. The two tables at the bottom show the correct outcome of the recursion.

separate tables for the profit sums and weight sums. We introduce two binary tables $g(j,\overline{w})$ and $h(j,\overline{p})$ for $j = 0,\ldots,n$, $\overline{w} = 0,\ldots,c$, and $\overline{p} = 0,\ldots,z^*$. The contents of the tables is defined by the following relation

$$\begin{array}{l} \text{an undominated state } (\overline{w},\overline{p}) \text{ exists in } z_j \quad \Rightarrow \\ g(j,\overline{w}) = 1 \quad \wedge \quad h(j,\overline{p}) = 1 \end{array} \tag{5.33}$$

Thus g is a table of the undominated weight sums and h is a table of the undominated profit sums. As we only need one bit to express the value of g and h, we may reduce the space complexity by storing W entries in each word, where W is the word size. The encoded representation $g(j,\overline{w}),h(j,\overline{p})$ can be transformed on-line to the ordinary representation $z_j(d)$ in linear time (and vice-versa). Hence, using the Bellman recursion (5.31) on the compact representation with on-line conversion of the data immediately leads to the improved space complexity $O(nm/\log m)$ where $m = \max\{b,z^*\}$. As z^* is not known in advance one may use U_{LP} for bounding z^*, noting that this will not affect the asymptotic complexity, since the bound can be derived in linear time and its value is at most twice the optimal solution.

The relation between the two tables $g(j,\overline{w})$ and $h(j,\overline{p})$ and the original table $z_j(d)$ is defined as follows:

Observation 5.2.1 *Assume that $(\overline{w}',\overline{p}')$ is the k-th undominated state in the list representation of $z_j(d)$. Then the k-th 1-bit in table $g(j,\overline{w})$ is found for $\overline{w} = \overline{w}'$ and the k-th 1-bit in table $h(j,\overline{p})$ is found for $\overline{p} = \overline{p}'$.*

To decrease the time complexity we need to prescribe an efficient way of getting from $g(j-1,\overline{w})$ to $g(j,\overline{w})$ and from $h(j-1,\overline{w})$ to $h(j,\overline{w})$. A naive approach is to use the same approach as for the subset sum problem independently on the weights and profits (i.e. tables g and h) which however is not a feasible approach as illustrated by Figure 5.4.

The example shows that the recursion must take both tables g and h into account. Hence to derive tables $g(j,\cdot)$ and $h(j,\cdot)$ from $g(j-1,\cdot)$ and $h(j-1,\cdot)$ we use the following approach. First we copy $g(j-1,\cdot)$ to a new table $g'(j-1,\cdot)$ which is

shifted right by w_i bits. In a similar way we copy $h(j-1,\cdot)$ to $h'(j-1,\cdot)$ shifted right by p_i bits. Now we need to merge the four tables $g(j-1,\cdot), g'(j-1,\cdot), h(j-1,\cdot), h'(j-1,\cdot)$ in a way which respects Observation 5.2.1. In the subset sum case we were so lucky that a simple machine operation was able to do the task. This is however not the case for (KP) hence we will implement the operation through a table-lookup. Since this is not a trivial task, we start by constructing an algorithm which can do the task for one bit. We may then generalize the algorithm to do the task for W bits.

Let us introduce the following variables:

k_g is the number of 1-bits met in table $g(j-1,\cdot)$.
k'_g is the number of 1-bits met in table $g'(j-1,\cdot)$.
k_h is the number of 1-bits met in table $h(j-1,\cdot)$.
k'_h is the number of 1-bits met in table $h'(j-1,\cdot)$.
ℓ_g is the current position in tables $g(j-1,\cdot)$ and $g'(j-1,\cdot)$.
ℓ_h is the current position in tables $h(j-1,\cdot)$ and $h'(j-1,\cdot)$.

To simplify the following algorithm we introduce a procedure Emit(e_g, e_h) which outputs the values e_g, e_h, increments ℓ_g or ℓ_h and adjusts k_g, k'_g, k_h, k'_h accordingly (see Figure 5.5 for details).

Algorithm Emit(e_g, e_h):

if $e_g \neq \perp$ then
  $g(j,\ell_g) := e_g$, $\ell_g := \ell_g + 1$,
  $k_g := k_g + g(j-1,\ell_g)$, $k'_g := k'_g + g'(j-1,\ell_g)$
if $e_h \neq \perp$ then
  $h(j,\ell_h) := e_h$, $\ell_h := \ell_h + 1$,
  $k_h := k_h + h(j-1,\ell_h)$, $k'_h := k'_h + h'(j-1,\ell_h)$

Fig. 5.5. The procedure Emit has as argument two values e_g and e_h which may have the values 0, 1, or $\perp$, where $\perp$ means that nothing is output. The values of e_g, e_h are output and all corresponding counters are increased accordingly.

The main algorithm for merging the four tables may now be described as shown in algorithm Wordmerge outlined in Figure 5.6. The algorithm maintains two windows to the tables, the first window is at position ℓ_g of the two tables $g(j-1,\cdot)$ and $g'(j-1,\cdot)$ while the second window is at position ℓ_h of the tables $h(j-1,\cdot)$ and $h'(j-1,\cdot)$. The two windows are repeatedly moved forward, so as to identify undominated pairs of states $(\overline{w},\overline{p})$.

Proposition 5.2.2 *Algorithm* Wordmerge *correctly merges the tables* g, g', h, h' *in time* $O(m)$ *where* $m := \max\{c, z^*\}$.

Algorithm Wordmerge:

0. $\ell_g := \ell_h := 0$, $k_g := g(0)$, $k'_g := g'(0)$, $k_h := h(0)$, $k'_h := h'(0)$
1. if $g(\ell_g) = 0$ and $g'(\ell_g) = 0$ then Emit$(0, \perp)$, goto 1
2a. if $g(\ell_g) = 1$ and $k_h < k_g$ then Emit$(\perp, 0)$, goto 1
2b. if $g'(\ell_g) = 1$ and $k'_h < k'_g$ then Emit$(\perp, 0)$, goto 1
3a. if $g(\ell_g) = 1$ and $h(\ell_h) = 1$ and $k_h = k_g$ then Emit$(1, 1)$, goto 1
3b. if $g(\ell_g) = 1$ and $h'(\ell_h) = 1$ and $k'_h = k'_g$ then Emit$(1, 1)$, goto 1
4a. if $g(\ell_g) = 1$ and $k_h \geq k_g$ then Emit$(0, \perp)$, goto 1
4b. if $g'(\ell_g) = 1$ and $k'_h \geq k'_g$ then Emit$(0, \perp)$, goto 1

Fig. 5.6. Algorithmic description of Wordmerge. All references in the figure are to tables $g(j-1,\cdot)$ and $h(j-1,\cdot)$ thus the subscript has been omitted to simplify the notation. To simplify the algorithm the stop criteria has been omitted, as it is a trivial task to ensure a proper termination.

	0	1	2	3	4
$g(1,\cdot)$	1	0	0	1	0
$g'(1,\cdot)$	0	1	0	0	1

	0	1	2	3	4
$h(1,\cdot)$	1	1	0	0	0
$h'(1,\cdot)$	0	0	0	1	1

ℓ_g	ℓ_h	step	action
0	0	3a	Emit$(1,1)$
1	1	2b	Emit$(\perp,0)$
1	2	2b	Emit$(\perp,0)$
1	3	3b	Emit$(1,1)$
2	4	1	Emit$(0,\perp)$
3	4	4a	Emit$(0,\perp)$
4	4	3b	Emit$(1,1)$

Fig. 5.7. Referring to the instance from Figure 5.4 the following tables illustrate how $g(2,\cdot)$ and $h(2,\cdot)$ are obtained using algorithm Wordmerge. Table $g'(1,\cdot)$ is obtained by shifting $g(1,\cdot)$ right by $w_2 = 1$ while table $h'(1,\cdot)$ is obtained by shifting $h(1,\cdot)$ right by $p_2 = 3$. The bottom table shows each step of algorithm Wordmerge and the corresponding emitted values.

Proof. To start with the time complexity, we notice that each iteration of the main loop emits at least one bit and thus increments either ℓ_g or ℓ_h. As $\ell_g \leq c$ and $\ell_h \leq z$ we immediately get the stated bound.

To prove the correctness we should recapitulate the definition of states in Observation 5.2.1. Thus **Step 1** is searching forward in the weight sum table g or g' until the weight-part of the next state is found. In Steps 2–4 we have identified this weight-part of the next state, and for symmetry reasons we may assume that $g(\ell_g) = 1$. Now there are three possibilities:

$k_h < k_g$ **(Step 2)** We have met more bits in table g than in table h thus we need to search forward to find the corresponding entry in h table to define a state $(\ell_g, \overline{p})$ where $\overline{p} > \ell_h$. Thus it seems safe to move forward in table h, but there is a

catch: It could be that (ℓ_g, ℓ_h) defined a state in table g', h'. If this was the case, the state (ℓ_g, ℓ_h) would be dominated by state $(\ell_g, \overline{p})$ and thus it is indeed safe to move forward in table h.

$k_h = k_g$ **(Step 3)** In this case we have met a similar amount of bits in tables g and h thus if $g(\ell_g) = h(\ell_h) = 1$ then the pair (ℓ_g, ℓ_h) defines a state. The state (ℓ_g, ℓ_h) cannot be dominated since this was already checked in Step 2.

$k_h > k_g$ **(Step 4)** In this case we have met more bits in table h than in table g thus we need to increase the position in table g to find a pair $(\overline{w}, \ell_h)$ which make a state. As all cases of dominance have been tested in the previous steps it is safe to move forward in table g.

□

As all decisions are based on the truth value of $\text{sign}(k_h - k_g)$, $\text{sign}(k'_h - k'_g)$ and the bits $g(j-1, \ell_g)$, $g'(j-1, \ell_g)$, $h(j-1, \ell_h)$, $h'(j-1, \ell_h)$, the whole process can be controlled by a single action table. Each step of algorithm Wordmerge consists of a single lookup in the action table which has dimension $3 \cdot 3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = (2+1)^2 2^4$. The output of the table will consist of a pair of bits stating the emitted values (e_g, e_h).

Assume that a new action table was obtained as a product of α times the original action table. In this way we would be able to process up to α bits in one single table lookup, thus getting an improved time and space complexity.

Proposition 5.2.3 *Algorithm* Wordmerge *controlled by an action table for* $\alpha = W/10$ *bits runs in* $O(n + m/\log m)$ *time and space where* $m := \max\{c, z^*\}$.

Proof. The correctness of the algorithm is obvious as the new action table is the product of α of the original action tables.

To prove the time complexity we notice that in each iteration we are able to process α bits, thus the time complexity is $O(m/\alpha)$. Due to assumption (5.32) we know that the word size corresponds to the size of the coefficients, i.e. that W is in $\Theta(\log m)$ and thus the running time is

$$O(m/\alpha) = O(m/W) = O(m/\log m). \tag{5.34}$$

The space complexity is obtained as follows: The action depends on the difference between the number of bits met in table g and h i.e. whether

$$k_h - k_g \leq -\alpha, \quad \text{or} \quad k_h - k_g = \{-\alpha + 1, \ldots, \alpha - 1\}, \quad \text{or} \quad k_h - k_g \geq \alpha.$$

Thus the number of distinct values is $2\alpha + 1$. The same applies for the value of $k'_h - k'_g$. Moreover, the action depends on the entries

$$g(i-1, \ell_g), g(i-1, \ell_g + 1), \ldots, g(i-1, \ell_g + \alpha - 1),$$

which can have 2^α different values. The same holds for $g'(i-1, \ell_g)$, $h(i-1, \ell_h)$ and $h'(i-1, \ell_h)$. In total the action table will have size $(2\alpha+1)^2 2^{4\alpha}$ i.e. it will use

$O(\alpha^2 2^{4\alpha})$ space. Now since $\alpha^2 \leq 2^\alpha$ for $\alpha \geq 2$, and since $\alpha = W/10$ it follows that $\alpha^2 2^{4\alpha} \leq 2^{5\alpha}$ and hence we get the space complexity

$$O(2^{W/2}). \tag{5.35}$$

The space consumption depends on the size of the action table and of the dynamic programming table. Due to assumption (5.32) we have W is in $\Theta(\log m)$ and thus the space complexity becomes

$$\begin{aligned} O(2^{W/2} + m/\log m) &= O(\sqrt{2^W} + m/\log m) \\ &= O(\sqrt{m} + m/\log m) \\ &= O(m/\log m). \end{aligned} \tag{5.36}$$

Since we also need to read and store the n given items, we get the stated time and space complexities. □

Using algorithm Wordmerge for all items $1,\ldots,n$ we immediately get the following result shown by Pisinger [392]:

Corollary 5.2.1. *The knapsack problem can be solved in $O(nm/\log m)$ time and $O(n + m/\log m)$ space where $m = \max\{c, z^*\}$.*

It should be pointed out that the word RAM model of computation allows any (fixed) number of constants, the so-called *native constants*. Since the action table does not depend on the given instance, it can be stored as a part of the algorithm, and hence is to be be considered as a table of native constants. This means that we may use α equal to the word size W of the processor without affecting the the worst-case time or space complexity.

5.2.2 Horowitz and Sahni Decomposition

We will only point out here the method by Horowitz and Sahni [238] which is based on the subdivision of the original problem of n variables into two subproblems of size $n/2$. For each subproblem a classical dynamic programming recursion is used. Finally, the two resulting dynamic programming arrays can be easily combined. The running time complexity of this approach is $O(2^{n/2})$ as well as the pseudopolynomial bound $O(nc)$. Thus, for difficult problems the recursion improves by a square-root over a complete enumeration of magnitude $O(2^n)$.

We refer also to Section 3.3 for a further discussion.

5.3 Primal-Dual Dynamic Programming Algorithms

The Bellman recursion (5.31) builds the optimal solution from scratch by repeatedly adding a new item to the problem. A different *primal-dual recursion* was presented

by Pisinger [387]. Although the recursion can be used to solve the *all-capacities* variant of (KP), it is mainly advantageous to use when solving the (KP) for a specific value of c.

The primal-dual recursion is based on the observation that generally only few items around the split item s need to be changed from their LP-optimal values in order to obtain the IP-optimal values. Thus, assume that the items are ordered according to (5.7), and let $z_{a,b}(d)$ for $a = 1,\dots,s$ and $b = s-1,\dots,n$ and $d = 0,\dots,2c$ be an optimal solution to the problem:

$$z_{a,b}(d) = \sum_{j=1}^{a-1} p_j + \max\left\{ \sum_{j=a}^{b} p_j x_j \;\middle|\; \begin{array}{l} \sum_{j=a}^{b} w_j x_j \le d - \sum_{j=1}^{a-1} w_j, \\ x_j \in \{0,1\}, j = a,\dots,b \end{array} \right\}. \tag{5.37}$$

As can be seen, $z_{a,b}(d)$ is an optimal solution to the knapsack problem defined on items $j = a,\dots,b$ where items $j < a$ have been fixed to 1, and items $j > b$ have been fixed to 0.

The following two recursions are used

$$z_{a,b}(d) = \begin{cases} z_{a,b-1}(d) & \text{if } d - w_b < 0, \\ \max\{z_{a,b-1}(d), z_{a,b-1}(d-w_b) + p_b\} & \text{if } d - w_b \ge 0, \end{cases} \tag{5.38}$$

$$z_{a,b}(d) = \begin{cases} z_{a+1,b}(d) & \text{if } d + w_a > 2c, \\ \max\{z_{a+1,b}(d), z_{a+1,b}(d+w_a) - p_a\} & \text{if } d + w_a \le 2c. \end{cases} \tag{5.39}$$

The first recursion (5.38) relates to the possible insertion of item b into the knapsack, while (5.39) relates to the possible removal of item a from the knapsack. See Section 4.1.2 for a similar recursion for the subset sum problem. Initially we set $z_{s,s-1}(d) = -\infty$ for $d = 0,\dots,\hat{w}-1$ and $z_{s,s-1}(d) = \overline{p}$ for $d = \hat{w},\dots,2c$, where $\hat{w}$ is the weight of the split solution defined in (5.10). Then we alternate between the two recursions (5.38) and (5.39) evaluating $z_{s,s-1}, z_{s,s}, z_{s-1,s}, z_{s-1,s+1},\dots$ until we reach $z_{1,n}$. The optimal solution value is then given by $z_{1,n}(c)$. Intuitively speaking, the enumeration starts at $(a,b) = (s,s-1)$ and continues by either removing an item $a < s$ from the knapsack, or inserting an item $b \ge s$ into the knapsack.

The running time of the algorithm is $O(nc)$ which is the same as that of DP-2 for the Bellman recursion. The benefits of this approach become obvious when combining dynamic programming with upper bounds as described in Section 3.5. Pisinger [387] used the bound U_0 to fathom unpromising states, terminating the dynamic programming recursion when all states had been fathomed. Computational experiments showed that in practice only a small subset of items around the split item s need to be considered. As the number of states typically is far smaller than $2c$ and the number of items considered is far smaller than n, the time bound of $O(nc)$ is very pessimistic.

5.3.1 Balanced Dynamic Programming

A different primal-dual dynamic programming algorithm can be obtained by using the concept of balancing as described in Section 3.6. Assume that the items are ordered according to decreasing efficiencies (5.7).

As a start we define a table $g(b,\overline{w},\overline{p})$ for indices $b = s-1,\dots,n$. The weight sum $\overline{w}$ should be in the interval $c - w_{\max} < \overline{w} \le c + w_{\max}$, and the profit sum $\overline{p}$ should be in the interval $\alpha(\overline{w}),\dots,\beta(\overline{w})$ where the two functions α and β will be described in the sequel. The entries of table $g(b,\overline{w},\overline{p})$ are defined by

$$g(b,\overline{w},\overline{p}) := \max_{a=1,\dots,b+1} \left\{ a \;\middle|\; \begin{array}{l} \text{a balanced filling } x \text{ exists with} \\ \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{b} w_j x_j = \overline{w}, \\ \sum_{j=1}^{a-1} p_j + \sum_{j=a}^{b} p_j x_j = \overline{p}, \\ x_j \in \{0,1\},\ j = a,\dots,b \end{array} \right\} \tag{5.40}$$

An entry $g(b,\overline{w},\overline{p}) = a$ means that there exists a balanced solution with weight sum $\overline{w}$, profit sum $\overline{p}$ where all items $1,\dots,a-1$ have been selected, and some of the items $a,\dots,b$ may be in the knapsack. If no balanced solution with this property exists we set $g(b,\overline{w},\overline{p}) = 0$. The triple $(b,\overline{w},\overline{p})$ will also be referred to as a *state* in the dynamic programming. Having filled the table $g(b,\overline{w},\overline{p})$, an optimal solution z^* to (KP) is found as the largest value of $\overline{p}$ for which $g(n,\overline{w},\overline{p}) \neq 0$, considering all values of $\overline{w} = c - w_{\max} + 1,\dots,c$.

For $b = s-1$ only one value of $g(b,\overline{w},\overline{p})$ is positive, namely $g(s-1,\hat{w},\hat{p}) = s$, since only the split solution initially is balanced.

As a consequence of Observation 3.6.3 the balanced solutions will satisfy that $c - w_{\max} < \overline{w} \le c + w_{\max}$. To derive a similar bound on the profit sums we apply bound U_1 given by (5.9). As an upper bound on a state $(b,\overline{p},\overline{w})$ we use the Dembo and Hammer bound (5.29) given by

$$U(b,\overline{p},\overline{w}) = \overline{p} + \lfloor (c-\overline{w})\frac{p_s}{w_s} \rfloor. \tag{5.41}$$

Obviously, an upper bound on a state cannot be better than the upper bound on (KP) since we may have fixed some decision variables in the state, thus we have $U(b,\overline{p},\overline{w}) \le U_1$. By setting

$$\beta(\overline{w}) := U_1 - \lfloor (c-\overline{w})\frac{p_s}{w_s} \rfloor \tag{5.42}$$

we have that $\overline{p} \le \beta(\overline{w})$.

In addition, any state $(b,\overline{p},\overline{w})$ can be discarded if its upper bound is worse than a given incumbent solution value z^ℓ. Thus any state not reduced must satisfy $U(b,\overline{p},\overline{w}) = \overline{p} + \lfloor (c-\overline{w})p_s/w_s \rfloor \ge z^\ell + 1$. Setting

$$\alpha(\overline{w}) := z^\ell + 1 - \lfloor (c-\overline{w})\frac{p_s}{w_s} \rfloor \tag{5.43}$$

we observe that $\overline{p} \geq \alpha(\overline{w})$.

Hence, at any stage we have $\alpha(\overline{w}) \leq \overline{p} \leq \beta(\overline{w})$, so the number of profit sums $\overline{p}$ corresponding to a weight sum $\overline{w}$ can at most be $\beta(\overline{w}) - \alpha(\overline{w}) + 1 = U_1 - z^\ell$. If we define

$$\Gamma := U_1 - z^\ell, \tag{5.44}$$

we notice that $\Gamma \leq p_{\max}$ since we may use the profit of the split solution $\hat{p}$ as incumbent solution z^ℓ, and thus $U_1 < \hat{p} + p_b$.

Algorithm Balknap:

```
1  for w̄ := c − w_max + 1 to c do
2    for p̄ := α(w̄) to β(w̄) do g(s − 1, w̄, p̄) := 0
3  for w̄ := c + 1 to c + w_max do
4    for p̄ := α(w̄) to β(w̄) do g(s − 1, w̄, p̄) := 1
5  g(s − 1, ŵ, p̂) := s
6  for b := s to n do
7    for w̄ := c − w_max + 1 to c + w_max do
8      for p̄ := α(w̄) to β(w̄) do g(b, w̄, p̄) := g(b − 1, w̄, p̄)
9    for w̄ := c − w_max + 1 to c do
10     for p̄ := α(w̄) to β(w̄) do
11       w̄' := w̄ + w_b, p̄' := p̄ + p_b
12       g(b, w̄', p̄') := max{g(b, w̄', p̄'), g(b − 1, w̄, p̄)}
13   for w̄ := c + w_b downto c + 1 do
14     for p̄ := α(w̄) to β(w̄) do
15       for j := g(b − 1, w̄, p̄) to g(b, w̄, p̄) − 1 do
16         w̄' := w̄ − w_j, p̄' := p̄ − p_j
17         g(b, w̄', p̄') := max{g(b, w̄', p̄'), j}
```

Fig. 5.8. The Balknap algorithm is a based on balanced dynamic programming. Its overall structure is similar to the Balsub algorithm for the subset sum problem.

To calculate the table $g(b,\overline{w},\overline{p})$ we may now use the dynamic programming algorithm Balknap described in Figure 5.8. The algorithm is based on the principles described in Section 3.6, although we use an iterative and thus more efficient way of filling table $g(b,\overline{w},\overline{p})$. The following description relates to parts a) to f) described in Section 3.6.

The first lines 1–5 correspond to the initialization as described in Step a). Lines 6–17 correspond to the enumeration of all balanced solutions as described in Step b). We use table $g(b,\overline{w},\overline{p})$ to save the pool of open nodes as defined in Steps d) and f). The concept of dominance described in Step c) can easily be generalized to also handle the profit sums. If two states $(a,b,\overline{w},\overline{p})$ and $(a',b',\overline{w}',\overline{p}')$ satisfy

$$b = b' \text{ and } \overline{w} = \overline{w}' \text{ and } \overline{p} = \overline{p}' \text{ and } a \geq a', \tag{5.45}$$

then the first subproblem dominates the latter, and we may remove the latter. This means that we will only have one open subproblem for each value of b, $\overline{p}$, $\overline{w}$. The

evaluation order of the open problems is chosen to be for increasing values of $b = s, \ldots, n$, decreasing values of $\overline{w}$ and increasing values of $\overline{p}$. This choice corresponds to the main loop in lines 6–17. In lines 7–8 we copy the states at level $b-1$ to level b as described in the end of Step d). This corresponds to line 7 in Figure 3.9 as we do not add item b to the knapsack. In lines 9–12 we add item b to all states with $\overline{w} \leq c$ which corresponds to line 8 in Figure 3.9. Finally in lines 13–17 we remove one or more items from the knapsack until all states have weight sum $\overline{w} \leq c$. This corresponds to repeated executions of lines 8–10 in Figure 3.9. As in line 10 of Figure 3.9 we only need to consider the removal of an item with index not less than $g(b-1, \overline{p}, \overline{w})$, as otherwise the removal has already been performed once before (see Step b), final note).

The time and space complexity is $O(nw_{\max}\Gamma)$ which easily may be verified using the same arguments as in Step e) of Section 3.6. Since $\Gamma \leq p_{\max}$ the complexity is bounded by the term $O(nw_{\max}p_{\max})$ which is linear provided that the magnitude of the weights and profits is bounded by a constant. As $\Gamma = U_1 - z^{\ell}$ we may improve the practical running times by providing upper and lower bounds of good quality. In Pisinger [387, page 86] it is shown that the gap $\Gamma = U_1 - z$ is relative small (typically smaller than 10) for large-sized instances from literature, thus the running time of Balknap for these instances will be about an order of magnitude larger than that of Balsub described in Section 4.1.5.

Note that a complete ordering of the items according to decreasing efficiencies is not necessary, as we only need to know the split item s for deriving upper and lower bounds, hence this part of the algorithm only takes $O(n)$ time, as shown in Section 3.1.

5.4 The Core Concept

Experimental evidence has shown for a large variety of test instances that only a relatively small subset of items is crucial for the exact structure of the optimal solution vector, in particular if the number of items is very large. Items with very high efficiencies will almost certainly be included in any good solution and also in the optimal solution set. On the other hand, items with very low efficiencies can be more or less eliminated from consideration right from the beginning since they will not be chosen in an optimal solution. Hence we are left over with the items of "medium efficiency".

Looking back at the LP-relaxation of (KP) it seems natural that the efficiency of the split item s may be taken as an orientation of what a medium efficiency might be. It is difficult to decide whether to pack items which have efficiency close to p_s/w_s. Such items are said to be the *core*.

Balas and Zemel [22] presented the following precise definition of the core of a knapsack problem which is however based on the knowledge of an optimal solution.

Assume that the items are ordered according to decreasing efficiencies as in (5.7) and let an optimal solution vector be given by x^*. Define

$$a := \min\{j \mid x_j^* = 0\}, \qquad b := \max\{j \mid x_j^* = 1\}. \tag{5.46}$$

If we define $\tilde{p} = \sum_{j=1}^{a-1} p_j$ and $\tilde{w} = \sum_{j=1}^{a-1} w_j$ then the *core* is given by the items in the interval $C = \{a, \ldots, b\}$ and the *core problem* is defined as

$$\begin{aligned} (\mathrm{KP}_C) \qquad & \text{maximize} \sum_{j \in C} p_j x_j + \tilde{p} \\ & \text{subject to} \sum_{j \in C} w_j x_j \leq c - \tilde{w}, \\ & x_j \in \{0,1\}, \qquad j \in C. \end{aligned} \tag{5.47}$$

For many classes of instances, with the so-called *strongly correlated knapsack problems* (see Section 5.5) as an important exception, the size of the core is only a small fraction of n. Hence, if we knew a priori the values of a and b we could easily solve the complete problem by setting $x_j^* = 1$ for $j = 1, \ldots, a-1$ and $x_j^* = 0$ for $j = b+1, \ldots, n$ and simply solving the core problem through branch-and-bound or dynamic programming.

There are several advantages of solving a core problem. First of all, the sorting of all items is avoided as only the items inside the core need to be considered. The sorting time of $O(n \log n)$ frequently dominates the solution time for uncorrelated large-sized instances, and thus a reduction of the sorting time will have a significant influence on the total solution time. Another motivation may be to focus the search on the most interesting items. Without solving a core, a depth-first branch-and-bound will typically have a very deep search tree, although good solutions seldom are found in the bottom of the search tree due to the small efficiency of the items. Thus focusing the search on a core, will normally mean that incumbent solutions of high quality are found more quickly. Finally, solving a core problem means that a large part of the variables outside the core can be fixed at their optimal value by a reduction algorithm as described in the previous section.

Pisinger [385] observed that there are also disadvantages of solving a core problem. Experimental evidence showed that in some cases the above definition of a core results in many items with similar or proportional weights. This degeneration of the core results in a difficult problem for branch-and-bound algorithms, since with many proportional weights it may be impossible to obtain a solution which fills the knapsack. Lacking a good lower bound, many algorithms get stuck in the enumeration of the core, although the inclusion of some small items outside the core could have made the problem easy to solve.

Computational comparisons reported in [385] showed that branch-and-bound algorithms are sensitive to the choice of the core, while dynamic programming algorithms in general are not affected in their performance, as similar weights are handled efficiently by dominance criteria.

5.4.1 Finding a Core

The definition of the core C is based on the knowledge of the optimal solution vector x^*. Since this knowledge is obviously not present, most algorithms rely on an approximated core, basically choosing 2δ items around the split item. Algorithms for finding a core were presented by e.g. Balas and Zemel [22], and Martello and Toth [333], although these algorithms only found a core of approximately the desired size. Pisinger [381] proposed a modification of the Quicksort algorithm (Hoare [231]) which determines the split item s and sorts 2δ items around s. The returned core is given as $C = \{a,\ldots,b\} = \{s-\delta,\ldots,s+\delta\}$. As in the ordinary quicksort algorithm the Find-Core algorithm partitions an interval $\{a,\ldots,b\}$ into two sets around the median γ of the efficiencies, such that $p_j/w_j \geq \gamma$ in the left interval, and $p_j/w_j \leq \gamma$ in the right interval. In the recursive step, the interval containing the split item s is always partitioned first. This means that s is well-defined when the second interval is considered upon backtracking, and thus it is trivial to check whether the interval is within the wanted core size $C = \{s-\delta,\ldots,s+\delta\}$.

Algorithm Find-Core$(a,b,\overline{w})$:

$\{a,\ldots,b\}$ *is an interval to be partitioned, and* $\overline{w} = \sum_{j=1}^{a-1} w_j$

1. Derive the median γ of the efficiencies and partition $\{a,\ldots,b\}$ into two intervals $\{a,\ldots,k-1\}$ and $\{k,\ldots,b\}$ satisfying that $p_j/w_j \geq \gamma$ for $j \in \{a,\ldots,k-1\}$ and $p_j/w_j \leq \gamma$ for $j \in \{k,\ldots,b\}$.
   Set $\tilde{w} := \sum_{j=1}^{k-1} w_j$.
2. If $\tilde{w} \leq c$ and $c < \tilde{w} + w_k$ then k is the split item hence set $s := k$.
3. If $\tilde{w} > c$ then goto Step 4
   else goto Step 5.
4. Call Find $-$ Core$(a,k-1,\overline{w})$.
   If $k \leq s+\delta$ then call Find $-$ Core$(m,b,\tilde{w})$
   else push interval $[k,b]$ to the stack L.
   Return.
5. Call Find $-$ Core$(k,b,\tilde{w})$.
   If $k-1 \geq s-\delta$ then call Find $-$ Core$(a,k-1,\overline{w})$
   else push interval $[a,k-1]$ to the stack H.
   Return.

Fig. 5.9. The Find-Core algorithm repeatedly partitions the current interval of items into two equally sized sets, and discards one of the sets as it cannot contain the split item s by looking at the overall weight sum of the included items. Discarded sets are pushed to two stacks H and L, where H contains the sets of items having a *higher* efficiency, and L contains the sets of items having a *lower* efficiency than the split item s.

The algorithm is called Find-Core$(1,n,0)$, and the two stacks H and L should be empty before the main call. Upon completion s will be the split item, and items in $C = \{s-\delta,\ldots,s+\delta\}$ will be sorted. All items $j < s-\delta$ have an efficiency larger than that of the items in the core, and items $j > s+\delta$ have smaller efficiency.

Since in each call to Find-Core we split the current interval of items $\{a,\ldots,b\}$ into equally large intervals, and only continue working on one of the intervals, we observe that

Lemma 5.4.1 *The algorithm* Find-Core *finds a core* $C = \{s-\delta,\ldots,s+\delta\}$ *in* $O(n)$ *time.*

As already noticed, the Find-Core algorithm is a normal sorting algorithm apart from the fact that only intervals containing the split item are partitioned further. Thus all discarded intervals represent a partial ordering of the items according to the efficiencies which may be of use to the following enumerative algorithm. Pisinger [381] hence proposed to push the delimiters of discarded intervals into two stacks H and L as sketched in the algorithm.

Several proposals have been made as to how large a core should be chosen. Balas and Zemel [22] argued that the size of the core is constant $|C| = 25$, Martello and Toth [333] chose $|C| = \sqrt{n}$, while Martello and Toth [336] used $|C| = 2\sqrt{n}$. This seemingly discrepancy in the core size relates to different interpretations of a core problem. Assume that an algorithm is able to guess the minimal core $C = \{a,\ldots,b\}$ as defined by (5.46). Even if the solution of the core problem finds the optimal solution value z^*, a subsequent reduction algorithm will typically not be able to fix all variables at their optimal value and hence optimality of the found solution cannot be proved. This motivates to use a larger core than given by (5.46) in order to prove optimality of the core solution.

n	un-correlated	weakly correl.	strongly correl.	inv.strongly correl.	almost str. correl.	subset sum
50	3	8	12	9	12	15
100	5	12	13	12	20	14
200	7	14	18	17	26	13
500	8	17	25	23	41	13
1000	11	17	36	32	59	13
2000	13	20	44	40	72	13
5000	14	21	79	62	97	13
10000	17	25	104	88	106	13

Table 5.1. Minimal core size (number of items), average of 100 instances.

Experimental investigations of the minimal core size according to definition (5.46) were reported by Pisinger [383] for various types of knapsack problems. The exact definition of the different problem types will be described later in Section 5.5. Table 5.1 gives the average value over 100 instances for every problem type, together with some computational experiments. The item weights w_j were uniformly distributed in $[1,1000]$. Note that the actual number of items which are considered by a branch-and-bound algorithm to prove optimality of the solution is much higher.

Hence, the size of a core in a practical algorithm must be larger than the values reported in Table 5.1. Except for extremely unfavorable problems like strongly correlated instances, choosing $|C| = 50$ should be reasonable for most applications of the core concept.

The size of the minimal core was studied theoretically by Goldberg and Marchetti-Spaccamela [186]. They showed that the core size will grow logarithmically in expectation with increasing problem size. For more details, see Section 14.2.

5.4.2 Core Algorithms

There are two main strategies to exploit the core concept in an exact algorithm for (KP) depending on how to manage the core. The first approach selects a *fixed core* at the beginning whereas the second variant operates on an *expanding core*.

Fixed-Core Algorithms. Algorithms which solve (KP) with a fixed core have been presented by Balas and Zemel [22], Fayard and Plateau [138], and Martello and Toth [333, 336]. We will here describe the MT2 algorithm [333] in more details while deviations to the other algorithms will be be discussed in the sequel. The basic framework of algorithm MT2 may be sketched as shown in Figure 5.10.

Algorithm MT2:

1. Determine an approximate core $C = \{s-\delta, \ldots, s+\delta\}$.
   Sort the items in the core.
   Compute the optimal solution z_C^* of the core problem using the MT1 algorithm described in Section 5.1.4
2. Derive the upper bound U_C given by (5.13).
   If $U_C \leq \sum_{j=1}^{s-\delta-1} p_j + z_C^*$ then the core solution is optimal, stop.
3. Reduce variables as described in Section 5.1.3.
   If all x_j, $j \notin C$, are fixed at their optimal value then the core solution is optimal, stop.
4. Sort the variables not fixed at their optimal value.
   Solve the remaining problem to optimality.

Fig. 5.10. The MT2 algorithm solves a core problem, reduces the items, and finally solve the reduced problem.

The core size in Step 1 is chosen as $|C| = \sqrt{n}$ for $n \geq 100$ and $|C| = n$ for smaller instances, meaning that for these small instances the whole problem is solved in Step 1. To solve the core problem in Step 1 and the remaining problem in Step 4 to optimality, Martello and Toth [333] applied the MT1 branch-and-bound algorithm, described in Section 5.1.4. In the reduction part of Step 3, the bound U_2 is used to reduce variables outside the core.

The algorithm by Balas and Zemel [22] also follows the above framework but differs in choosing a core size of $|C| = 50$ and finding only an approximate solution to the core problem in Step 1. Also the upper bound in Step 2 and the enumeration in Step 4 are done by different subprocedures.

The Fayard and Plateau algorithm [138] is not strictly a core algorithm as the core only contains the split item s. Hence the initial solution z_C in Step 1 becomes the split solution where additional small items j with $\frac{p_j}{w_j} \leq \frac{p_s}{w_s}$ are possibly added to the knapsack. The bound U_C in Step 2 becomes the U_1 bound. In Step 3 the Dembo and Hammer bound (5.29) is applied. For the enumeration in Step 4, Fayard and Plateau used a branch-and-bound algorithm specialized for this problem.

A detailed description of all the above algorithms can be found in Martello and Toth [335] including some computational comparisons of the algorithms.

Martello and Toth [336] proposed a special variant of MT2 which was developed to deal with hard knapsack problems, i.e. problems where the structure of the data causes severe problems for the above algorithms. In order to avoid the problems discussed at the end of Section 5.4, the resulting algorithm MThard makes use of a larger core size in Step 1, namely $|C| = 2\sqrt{n}$ for $n \geq 100$. Furthermore, an upper limit on the number of the nodes considered in the branch-and-bound algorithm was imposed in Step 1. The last step of the algorithm (solving the remaining problem to optimality) is split into two parts: First the problem is solved through the ordinary MT1 algorithm with a limit on the number of branch-and-bound nodes investigated. If optimality of the obtained solution can be proved, the algorithm terminates. Otherwise, a new branch-and-bound algorithm is applied where the bound U_4 given by (5.20) and a similar bound based on minimum cardinality is applied. To speed up the solution process, a table of optimal sub-solutions is constructed for small values of the capacity, making it possible for the branch-and-bound algorithm to choose the proper sub-solution without additional branching. The table of optimal sub-solutions is constructed through dynamic programming.

Expanding Core Algorithms. Realizing that the core size is difficult to estimate in advance, Pisinger [381] proposed to use an expanding core algorithm, which simply starts with a core consisting of the split item only, and then adds more items to the core when needed. Using branch-and-bound for the enumeration, one gets the Expknap algorithm outlined in Figure 5.11.

Since sorting and reduction are only done when needed, no more effort will be made than absolutely necessary. Also the reductions are tighter when postponed in this way, since the later a reduction test is performed in an enumeration the better lower bound will be available.

A disadvantage of the Expknap algorithm is that the enumeration may follow an unpromising branch to completion, thus in the worst-case extending the core with all items, even if the instance could be solved with a smaller core by following a different branch. This inadequate behavior is avoided by using dynamic programming since the breadth-first search ensures that all variations of the solution vector

Algorithm Expknap:

1. Find core $C = \{s\}$ through algorithm Find-Core pushing all discarded intervals into two stacks H and L containing respectively items with higher or lower efficiencies than that of the split item s.
2. Find a heuristic solution z using the forward and backward heuristics (5.24) and (5.25).
3. Run algorithm Primal-Dual-Branch.
   For each recursive call, test if the parameters a, b to Primal-Dual-Branch, are within the current core C.
   If this is not the case then the next interval I from the corresponding stack H or L is popped, and the items in I are reduced using bounds (5.29). The remaining items in I are sorted according to decreasing efficiencies, and added to the core C.

Fig. 5.11. The Expknap algorithm is based on the primal-dual branch-and-bound algorithm Primal-Dual-Branch. As the branching proceeds, items are gradually reduced and sorted when needed.

have been considered before a new item is added to the core. The resulting Minknap algorithm was presented by Pisinger [383] and is outlined in Figure 5.12.

Algorithm Minknap:

1. Find an initial core $C = \{s\}$ as in the Expknap algorithm.
2. Run the dynamic programming recursion alternately using recursion (5.38) to insert an item b and recursion (5.39) to remove an item a.
   For each step, check if a and b are within the current core C, otherwise pick the next interval from H or L, reduce and sort the items, finally adding the remaining items to the core.
   Before an item a or b is considered in the recursion, a reduction test is performed, using the enumerative upper bound U_C given by (5.13).
   The dynamic programming algorithm is combined with upper bound tests as described in Section 3.5, using upper bound U_0 to fathom unpromising states.
3. The process stops when all states in the dynamic programming have been fathomed due to an upper bound test, or all items $j \notin C$ have been fixed at their optimal value.

Fig. 5.12. The Minknap algorithm can be seen as a kind of breadth-first search variant of Expknap. The algorithm is based on primal-dual dynamic programming algorithm, combined with a lazy kind of reduction and sorting.

Due to the breadth first-search of the dynamic programming, Pisinger proved the following property of the Minknap algorithm.

Theorem 5.4.2 *The* Minknap *algorithm enumerates at most the smallest* symmetrical *core* $C = \{s,t\}$ *around* s *which can be solved by a* fixed-core algorithm.

The meaning of "smallest" in this context is, that one cannot find a smaller core, which has symmetrical size around the split item s, such that a fixed-core algorithm like MT2 will terminate before reaching the last enumeration in Step 4. Table 5.1 has been generated by use of this property.

Notice that all fixed-core algorithms described in Section 5.4.2 can be seen as special cases of the MT2 algorithm, hence it is sufficient to show that the Minknap algorithm will not enumerate a larger core than MT2.

Proof. The core is symmetrical around s since the Minknap algorithm always extends the core in a symmetrical way. Assume that $\{a,\ldots,b\}$ is the smallest symmetrical core which can be solved by the fixed-core algorithm MT2 in its Step 2 or 3.

We want to prove that Minknap also will terminate when reaching $C = \{a,\ldots,b\}$. Notice that for $C = \{a,\ldots,b\}$, the Minknap algorithm will achieve the same lower bound $z^{\ell} = \sum_{j=1}^{a-1} p_j + z_C^*$ as found in Step 1 of MT2.

If the MT2 algorithm terminates in Step 2, then

$$U_C \leq \sum_{j=1}^{a-1} p_j + z_C^*, \tag{5.48}$$

where U_C is given by (5.13). But this means that when the Minknap algorithm reaches the core $\{a,\ldots,b\}$ then all states $i \in X_C$ will be fathomed since Minknap is using dynamic programming with upper bound tests as described in Section 3.5. Thus algorithm Minknap will also terminate when reaching this core.

If the MT2 algorithm terminates in Step 3, then the reduction algorithm described in Section 5.1.3 fathomed all items, meaning that

$$\begin{aligned} U_j^0 &< z+1 \text{ for } j = 1,\ldots,s-1, \\ U_j^1 &< z+1 \text{ for } j = t+1,\ldots,n, \end{aligned} \tag{5.49}$$

where Martello and Toth used the bound U_2 for this reduction. But the enumerative bound U_C used in Minknap holds the Martello-Toth upper bound as a special case with a core $C = \{s\}$. This implies that all items $j = 1,\ldots,a-1$ and $j = b+1,\ldots,n$ will be fathomed by the enumerative bound test, and the Minknap algorithm will not enumerate any item outside $\{a,\ldots,b\}$.

□

5.4.3 Combining Dynamic Programming with Tight Bounds

The currently most successful algorithm was presented by Martello, Pisinger and Toth [321]. Learning from the shortfalls of the different algorithms developed be-

fore, the authors constructed an algorithm which can be seen as a combination of many different concepts and is hence called Combo. Since the algorithm is based on dynamic programming, it has the nice pseudopolynomial time bound of $O(nc)$, although in general this bound is very pessimistic as most instances are solved very quickly due to bounding of states and reduction of variables.

The enumeration part of Combo is based on the same dynamic programming algorithm as Minknap but additional techniques are gradually introduced if the problem seems to be hard to solve. The number of states in the dynamic programming gives a good indication of the hardness of a problem, hence the additional techniques are introduced if the number of states exceeds some given limit: M_1, M_2 and M_3 respectively. By using this cascade of techniques which are gradually taken into use, one gets the same fast solution times as Minknap for easy instances, while more expensive techniques are used for the difficult instances. The main algorithm may be outlined as illustrated in Figure 5.13.

Algorithm Combo(M_1, M_2, M_3):

1. The dynamic programming recursion starts with an initial core $C = \{q, q', r, r', s-1, s, t, t'\}$ of items which fit together well. The dynamic programming recursion is run as described in the Minknap algorithm.
2. If the number of states in the dynamic programming recursion grows beyond a given value M_1, the greatest common divisor d of the weights is derived: if $d \neq 1$, the capacity is decreased to $c = d\lfloor c/d \rfloor$.
3. If the number of states exceeds a given value $M_2 > M_1$, a minimum or maximum cardinality constraint is derived. The constraint is surrogate relaxed with the original weight constraint, and the relaxed problem is solved to optimality by means of Minknap.
4. If the number of states exceeds a given value $M_3 > M_2$, an attempt is made to improve the lower bound by pairing dynamic programming states with items outside the core.
5. If the previous attempts fail, the algorithm continues as the Minknap algorithm, now having improved upper and lower bounds.

Fig. 5.13. The combo algorithm is based on the same dynamic programming approach as Minknap, but a number of additional techniques for tightening the upper and lower bounds are introduced based on the hardness of the problem.

The three constants M_1, M_2 and M_3, which are used to determine the "hardness" of a given instance, were experimentally estimated as 1000, 2000 and 10000, respectively, although computational experiments showed that the values are not critical.

In Step 1 a core C containing up to eight items is constructed by use of some heuristic rules. As discussed in the end of Section 5.4 choosing a core of items with similar efficiency may in rare situations lead to difficult core problems. Hence an initial core

is constructed by first finding the split item s in $O(n)$ time using the Find-Core algorithm, selecting items $\{s-1, s\}$ for the core C supplemented with some additional items. The first two additional items, q and q', are found by using the forward and backward greedy heuristics (5.24) and (5.25): q is the item of highest profit which can be added to the split solution once item $s-1$ has been removed, while q' is the item of lowest profit which has to be removed from the split solution so that the split item s can be added. Next, t is the item of highest profit which can be added to the split solution, while t' is the item of lowest profit which has to be removed from the split solution so that items s and $s+1$ can be added. Finally items r and r' having the smallest and largest weight, are added to the core, to ensure some variation in the weights. The resulting core C is sorted according to decreasing efficiencies of the items, and a complete enumeration of these items is performed resulting in some initial states for the dynamic programming recursion. Subsequently the dynamic programming recursion is run as described in the Minknap algorithm.

In Step 2 rudimentary divisibility arguments are used to decrease the capacity whenever possible. Let d be the greatest common divisor of the weights $w_1, \ldots, w_n$. If $d \neq 1$, the capacity may be decreased to $c = d\lfloor c/d \rfloor$. Deriving d can be done in $O(n \log w_{\max})$ time using Euclid's algorithm (see e.g. Cormen et al. [92, Section 31]).

In Step 3 upper bounds are derived by imposing cardinality constraints of the form (5.14). For symmetry reasons, only maximum cardinality constraints are considered in the following. The cardinality constraint is surrogate relaxed with the original capacity constraint using multiplier $\mu \geq 0$. This leads to a new (KP) of the form:

$$
\begin{aligned}
S(Ck\text{KP}, \mu) \qquad & \text{maximize} \sum_{j=1}^{n} p_j x_j \\
& \text{subject to} \sum_{j=1}^{n} (w_j + \mu) x_j \leq c + k\mu, \\
& x_j \in \{0,1\}, \qquad j = 1, \ldots, n.
\end{aligned}
\tag{5.50}
$$

This corresponds to the integer solution of (5.21). By applying the same technique described for the latter problem, the optimal multiplier μ for the LP-relaxation of (5.50) can be found in time $O(n \log p_{\max} + n \log w_{\max})$ by means of binary search. Although the best multiplier μ for the LP relaxation of (5.50) is not necessarily optimal for the IP problem, it is of good quality in practice.

The new problem (5.50) is also a knapsack problem, hence it is solved by use of the Minknap algorithm. The transformed problem tends to be much easier to solve than (KP), since bounds from linear relaxation for this problem are very close to the optimal solution value. An optimal solution to the transformed IP-problem yields an upper bound on the original problem, and if the cardinality of the solution satisfies (5.14), i.e. if $\sum_{j=1}^{n} x_j \leq k$ in the optimal solution x to (5.50), the solution x is also feasible to the original problem having an objective value which matches the upper bound. Hence in such cases the problem is solved to optimality.

Finally in Step 4 the lower bound z^ℓ is attempted to be improved by pairing items $j \notin C$ with states in the dynamic programming table. Since the dynamic programming recursion is based on lists as described in Section 3.4 we may assume that at the present stage we have the list of states $L = \langle(\overline{w}_1, \overline{p}_1), (\overline{w}_2, \overline{p}_2), \ldots, (\overline{w}_m, \overline{p}_m)\rangle$.

For each item $j \notin C$ with efficiency larger than that of s, we find the state i with largest weight sum $\overline{w}_i$ such that $\overline{w}_i \leq c + w_j$. In this way we also obtain the largest profit sum $\overline{p}_i$ of feasible sub-solutions. This leads to the solution value $z_j = \overline{p}_i - p_j$. Similarly, for each item $j \notin C$ with efficiency smaller than that of s, we find the state i with the largest weight sum $\overline{w}_i$ such that $\overline{w}_i \leq c - w_j$. The corresponding objective value is $z_j = \overline{p}_i + p_j$. The lower bound is then improved as $z^\ell := \max\{z^\ell, \max_{j \notin C} z_j\}$.

Since the states are ordered by increasing weight and profit sums, the proper state can be found in time $O(\log M_3)$ through binary search. This gives a total time bound of $O(n \log M_3)$ for Step 4, where M_3 is a constant.

If the above techniques were not able to prove optimality of the lower bound z^ℓ, Step 5 continues with the Minknap algorithm, now having better upper and lower bounds available.

5.5 Computational Experiments

To give the reader some orientation about the performance of different exact knapsack algorithms we will conclude this chapter with the presentation of some computational experiments. Concerning the performance of various older algorithms the comparison with MT2 reported in [335] should be consulted.

To get a more nuanced picture of the algorithms we will consider several groups of randomly generated instances of (KP) which have been constructed to reflect special properties that may influence the solution process. In all instances the weights are uniformly distributed in a given interval with *data range* $R = 1000$ and 10000. The profits are expressed as a function of the weights, yielding the specific properties of each group. The instance groups are graphically illustrated in Figure 5.14.

- **Uncorrelated data instances**: p_j and w_j are chosen randomly in $[1, R]$. In these instances there is no correlation between the profit and weight of an item. Such instances illustrate those situations where it is reasonable to assume that the profit does not depend on the weight. Uncorrelated instances are generally easy to solve, as there is a large variation between the profits and weights, making it easy to fathom numerous variables by upper bound tests or by dominance relations.
- **Weakly correlated instances**: weights w_j are chosen randomly in $[1, R]$ and the profits p_j in $[w_j - R/10, w_j + R/10]$ such that $p_j \geq 1$. Despite their name, weakly correlated instances have a very high correlation between the profit and weight of an item. Typically the profit differs from the weight by only a few percent. Such instances are perhaps the most realistic in management, as it is well-known that

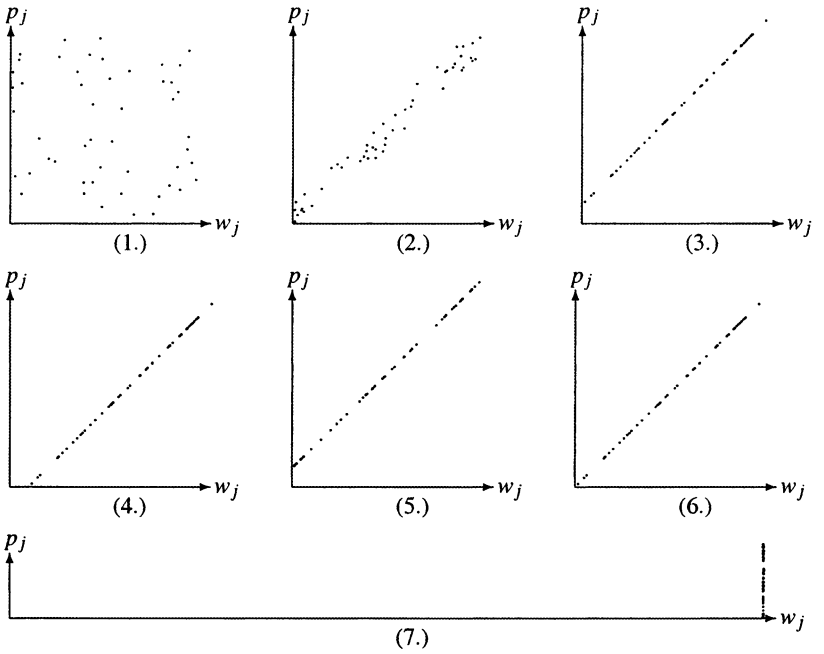


Fig. 5.14. Classical test instances. (1.) Uncorrelated instances. (2.) Weakly correlated instances. (3.) Strongly correlated instances. (4.) Inverse strongly correlated instances. (5.) Almost strongly correlated instances. (6.) Subset sum instances. (7.) Uncorrelated instances with similar weights. Note that instances (3.) and (5.) look very similar since the extra "noise" in almost strongly correlated instances is very small.

the return of an investment is generally proportional to the sum invested within some small variations.

- **Strongly correlated instances**: weights w_j are distributed in $[1,R]$ and $p_j = w_j + R/10$. Such instances correspond to a real-life situation where the return is proportional to the investment plus some fixed charge for each project. The strongly correlated instances are hard to solve for two reasons:
  a) The instances are *ill-conditioned* in the sense that there is a large gap between the continuous and integer solution of the problem.
  b) Sorting the items according to decreasing efficiencies correspond to a sorting according to the weights. Thus for any interval of the ordered items there is a small variation in the weights, making it difficult to satisfy the capacity constraint with equality.
- **Inverse strongly correlated instances**: profits p_j are distributed in $[1,R]$ and $w_j = p_j + R/10$. These instances are like strongly correlated instances, but the fixed charge is negative.
- **Almost strongly correlated instances**: weights w_j are distributed in $[1,R]$ and the profits p_j in $[w_j + R/10 - R/500,\ w_j + R/10 + R/500]$. These are a kind of

fixed-charge problem with some noise added. Thus they reflect the properties of both strongly and weakly correlated instances.

- **Subset sum instances**: weights w_j are randomly distributed in $[1,R]$ and $p_j = w_j$. These instances reflect the situation where the profit of each item is equal (or proportional) to the weight. Thus, our only goal is to obtain a filled knapsack. Subset sum instances are however difficult to solve as instances to (KP) because most of the considered upper bounds return the same trivial value c, and thus we cannot use bounding rules for cutting off branches before an optimal solution has been found.
- **Uncorrelated instances with similar weights**: weights w_j are distributed in $[100000, 100100]$ and the profits p_j in $[1, 1000]$.

	uncorr.		weak.corr		str.corr		inv.str.corr		al.str.corr		subs.sum		sim.w
$n \backslash R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.1	0.0	0.1	0.0	5.8	4.7	1.2	1.8	2.9	3.0	0.0	0.8	1.7
100	0.0	0.1	0.1	0.1	2239.9	2235.9	359.7	15083.3	582.7	1607.7	0.1	0.8	328.7
200	0.1	0.1	0.3	0.4	—	—	—	—	—	—	0.2	2.9	—
500	0.1	0.4	0.7	1.1	—	—	—	—	—	—	0.5	1.8	—
1000	0.3	1.0	0.8	2.5	—	—	—	—	—	—	0.4	2.2	—
2000	1.0	1.4	0.9	4.1	—	—	—	—	—	—	0.7	2.3	—
5000	2.3	3.8	1.5	9.5	—	—	—	—	—	—	0.9	3.0	—
10000	2.9	4.9	2.3	14.5	—	—	—	—	—	—	1.4	3.2	—

Table 5.2. Average solution times in milliseconds, MT2 (INTEL PENTIUM III, 933 MHz).

	uncorr.		weak.corr		str.corr		inv.str.corr		al.str.corr		subs.sum		sim.w
$n \backslash R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.0	0.0	0.0	0.0	10.7	6.0	2.1	10.0	6.5	3.3	0.1	0.7	2.6
100	0.1	0.0	0.2	0.2	1438.4	3795.8	3517.3	11232.4	3604.9	2452.2	0.0	0.8	2289.8
200	0.0	0.1	0.2	0.4	—	—	—	—	—	—	0.1	0.8	—
500	0.1	0.2	0.6	1.2	—	—	—	—	—	—	0.1	1.0	—
1000	0.5	0.1	0.4	2.9	—	—	—	—	—	—	0.0	0.9	—
2000	0.4	0.7	0.6	7.4	—	—	—	—	—	—	0.2	1.4	—
5000	1.0	2.3	0.7	21.1	—	—	—	—	—	—	0.3	1.5	—
10000	1.3	5.3	1.5	34.3	—	—	—	—	—	—	0.7	1.8	—

Table 5.3. Average solution times in milliseconds, Expknap (INTEL PENTIUM III, 933 MHz).

We will compare the performance of MT2, Expknap, Minknap, MThard and Combo. The behavior of the algorithms will be considered for different problem sizes n, different problem types, and two data ranges. For each instance type a series of $H = 100$ instances is performed. The capacity in each instance is chosen as $c = \frac{h}{H+1} \sum_{j=1}^{n} w_j$, for test instance number $h = 1, \ldots, H$.

	uncorr.		weak.corr		str.corr		inv.str.corr		al.str.corr		subs.sum		sim.w
$n \backslash R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.1	0.0	0.0	0.1	0.4	2.3	0.4	2.5	0.4	0.3	0.3	5.4	0.0
100	0.1	0.0	0.1	0.2	1.9	31.1	1.4	33.5	1.5	3.9	0.2	6.1	0.1
200	0.0	0.0	0.1	0.2	5.6	163.2	4.4	122.0	5.1	20.9	0.2	6.9	0.6
500	0.2	0.1	0.0	0.2	20.0	523.0	20.6	594.7	17.1	155.6	0.3	5.9	3.1
1000	0.1	0.1	0.2	0.9	53.9	1829.0	51.7	1655.1	43.6	625.0	0.1	6.0	12.1
2000	0.5	0.5	0.3	1.2	113.9	2899.1	141.6	3196.0	86.2	1863.0	0.3	6.7	40.7
5000	0.8	1.0	0.4	1.9	542.6	12403.1	454.0	12571.5	216.7	6103.8	0.3	7.5	150.5
10000	1.2	1.3	1.0	3.4	1301.9	26713.6	954.9	27972.2	250.5	11274.7	1.1	7.8	167.4

Table 5.4. Average solution times in milliseconds, Minknap (INTEL PENTIUM III, 933 MHz).

	uncorr.		weak.corr		str.corr		inv.str.corr		al.str.corr		subs.sum		sim.w
$n \backslash R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.1	0.0	0.0	0.1	0.7	1.1	0.6	0.6	1.2	5.5	0.1	0.7	0.5
100	0.1	0.0	0.1	0.1	1.4	2.0	1.1	1.8	3.5	25.5	0.1	0.7	1.1
200	0.1	0.1	0.5	0.4	8.0	9.1	4.0	5.2	11.5	60.1	0.3	2.2	5.4
500	0.4	0.5	1.0	1.0	18.9	18.9	11.0	12.4	20.5	131.6	0.4	1.8	12.4
1000	1.0	1.1	1.4	2.1	32.0	39.6	18.6	24.1	39.0	182.3	0.5	3.5	23.6
2000	1.7	1.8	1.6	4.4	42.5	66.4	26.2	33.7	63.5	182.5	0.4	2.0	40.1
5000	2.9	6.0	2.8	9.1	54.8	225.0	42.9	92.2	316.2	348.6	0.6	2.9	54.5
10000	4.9	10.4	3.2	17.9	92.7	847.6	64.3	338.5	—	—	1.4	147.9	76.6

Table 5.5. Average solution times in milliseconds, MThard (INTEL PENTIUM III, 933 MHz).

	uncorr.		weak.corr		str.corr		inv.str.corr		al.str.corr		subs.sum		sim.w
$n \backslash R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.0	0.0	0.1	0.0	0.2	1.4	0.3	1.3	0.2	0.4	0.2	4.1	0.1
100	0.1	0.0	0.1	0.1	0.8	4.2	0.7	4.6	0.8	1.4	0.1	3.6	0.2
200	0.1	0.2	0.3	0.2	1.7	5.5	1.2	5.7	1.4	3.7	0.0	5.9	0.5
500	0.1	0.3	0.2	0.2	1.3	8.3	1.8	5.6	1.7	2.0	0.0	5.3	2.4
1000	0.1	0.3	0.4	0.7	1.6	10.4	2.4	5.6	3.1	1.4	0.3	4.3	4.7
2000	0.0	0.6	0.4	1.2	2.8	5.3	3.0	7.6	2.2	2.5	0.3	5.2	6.2
5000	0.7	1.0	0.3	3.2	4.2	5.8	4.4	6.4	3.6	4.6	0.4	4.0	10.2
10000	1.6	2.0	0.9	4.8	9.0	8.7	8.8	11.3	8.3	9.6	0.7	3.0	13.1

Table 5.6. Average solution times in milliseconds, Combo (INTEL PENTIUM III, 933 MHz).

All tests were run on an INTEL PENTIUM III, 933 MHz, and a time limit of 1 hour was assigned to each instance type for all H instances. If any of the instances were not solved within the time limit, this is indicated by a dash in the table.

Tables 5.2 to 5.6 compare the solution times of the five algorithms. The simple branch-and-bound codes MT2 and Expknap, have the overall worst performance, although they are quite fast on easy instances like the uncorrelated and weakly correlated ones. Moreover they are the fastest codes for the subset sum instances. For

the different variants of strongly correlated instances, MT2 and Expknap are able to solve only tiny instances.

The dynamic programming algorithm Minknap has an overall stable performance, as it is able to solve all instances within reasonable time. It is the fastest code for uncorrelated and weakly correlated instances and it has almost the same good performance for subset sum instances as MT2. The strongly correlated instances take considerably more time to be solved but although Minknap uses only simple bounds from LP-relaxation, the pseudopolynomial time complexity gets it safely through these instances.

The MThard algorithm has a very good overall performance, as it is able to solve nearly all problems within seconds. The short solution times are mainly due to its cardinality bounds which make it possible to terminate the branching after having explored a small number of nodes. There are however some anomalous entries for large-sized almost strongly correlated problems, where the cardinality bounds somehow fail. Also for some large-sized subset sum problems MThard takes very long time.

The best performance is obtained with the Combo algorithm. This "hybrid" algorithm solves all the considered instances within 15 milliseconds on average and the running times have a very systematic growth rate with small variations. In particular, it is worth noting that the ratio between solving a difficult instance and an easy instance is within a factor of 10, thus showing that the additional work needed to derive tight bounds can be done very fast.

Based on the results in Table 5.6 one could draw the wrong conclusion that (KP) is easy to solve. One should however notice that we consider instances where all the coefficients are of moderate size. If a real-life instance has this property (or can be scaled down to satisfy the property without significantly affecting the solution), then the problems are indeed easy to solve. However, there still exist many applications where more or less intractable knapsack instances with very large coefficients occur.

5.5.1 Difficult Instances

There are two directions to follow when constructing difficult instances: One may either consider the traditional instances with larger coefficients. This will obviously make the dynamic programming algorithms run slower, but also the upper bounds get weakened since the gap to the optimal solution is scaled and cannot be closed by simply rounding down the upper bound to the nearest smaller integer. A second direction to follow is to construct instances where the coefficients are of moderate size, but where the upper bounds presented in Section 5.1.1 have a bad performance.

In the following we will give examples on these difficult instances as presented by Pisinger [393]. Additional computational experiments may be found in the quoted paper.

	uncorr.			weak.corr			str.corr			inv.str.corr			al.str.corr			subs.sum			sim.w
$n \backslash R$	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^8
50	0.1	0.0	0.0	0.1	0.1	0.1	15	6	12	6	5	5	7	3	3	6	58	448	3
100	0.1	0.1	0.1	0.1	0.2	0.2	13690	2889	7089	2811	1224	4204	1478	3543	1127	8	58	678	2605
200	0.1	0.1	0.1	0.4	0.4	0.4	—	—	—	—	—	—	—	—	—	8	83	606	—
500	0.2	0.2	0.2	1.6	1.4	0.1	—	—	—	—	—	—	—	—	—	8	70	576	—
1000	0.6	0.6	0.7	3.8	4.3	1.7	—	—	—	—	—	—	—	—	—	11	64	779	—
2000	1.5	1.1	0.8	10.3	11.7	5.5	—	—	—	—	—	—	—	—	—	14	67	628	—
5000	3.3	1.7	1.6	36.5	1.6	1.5	—	—	—	—	—	—	—	—	—	16	77	660	—
10000	9.7	8.3	3.1	106.7	74.4	3.9	—	—	—	—	—	—	—	—	—	25	88	558	—

Table 5.7. Average solution times in milliseconds, Expknap, for large range instances (INTEL PENTIUM III, 933 MHz).

	uncorr.			weak.corr			str.corr			inv.str.corr			al.str.corr			subs.sum			sim.w
$n \backslash R$	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^8
50	0.1	0.1	0.0	0.2	0.1	0.1	23	26	40	11	21	15	3	2	2	84	—	—	0
100	0.1	0.0	0.1	0.2	0.2	0.2	242	1183	—	172	776	—	13	16	12	87	—	—	1
200	0.2	0.1	0.2	0.4	0.6	0.4	1292	—	—	1295	—	—	66	80	66	96	—	—	2
500	0.2	0.3	0.3	1.1	1.3	0.3	5114	—	68	5119	—	0	398	463	1	97	—	2376	18
1000	0.5	0.8	0.6	2.5	2.7	1.6	—	—	—	14716	—	—	1124	1410	183	91	—	—	72
2000	1.6	1.1	1.0	5.0	5.1	2.8	—	—	—	31094	—	0	3532	4463	495	100	—	—	1
5000	3.4	1.9	1.8	12.5	2.0	2.2	—	—	—	—	1	—	12629	8	7	105	402	4656	2
10000	6.5	5.7	4.1	24.0	17.5	4.0	—	—	—	—	—	—	25760	4938	86	122	—	—	5782

Table 5.8. Average solution times in milliseconds, Minknap, for large range instances (INTEL PENTIUM III, 933 MHz).

	uncorr.			weak.corr			str.corr			inv.str.corr			al.str.corr			subs.sum			sim.w
$n \backslash R$	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^5	10^6	10^7	10^8
50	0.1	0.1	0.1	0.2	0.2	0.2	14	81	175	15	—	—	2	2	2	39	602	—	0
100	0.0	0.1	0.1	0.2	0.2	0.4	160	985	—	19	454	—	8	10	7	23	605	—	1
200	0.1	0.2	0.2	0.6	0.6	0.5	1079	8706	—	29	751	—	30	36	20	15	346	—	2
500	0.3	0.5	0.3	1.2	1.6	0.4	4225	—	72	19	—	0	245	330	1	17	208	15	11
1000	0.7	0.7	0.6	2.5	2.8	1.6	—	—	—	19	446	—	284	485	144	16	145	—	15
2000	1.6	1.4	1.0	5.3	5.1	3.1	—	—	—	21	77	0	462	1114	140	16	62	—	1
5000	2.9	2.0	1.7	13.6	1.9	2.3	—	524	4238	34	1	—	982	8	6	17	16	16	2
10000	7.4	6.1	3.4	27.2	18.8	3.5	—	—	4948	47	56	—	4476	340	13	20	21	21	90

Table 5.9. Average solution times in milliseconds, Combo, for large range instances (INTEL PENTIUM III, 933 MHz).

5.5.2 Difficult Instances with Large Coefficients

Our first attempt to construct difficult instances is to use the "standard" instances from Section 5.5 for increasing data range R. The motivation being, that we know that the magnitude of the weights strictly affect the computational complexity of dynamic programming algorithms, hence in this way the effect of upper bound test, reduction, and early termination becomes more clear.

For the following experiments, the codes were compiled using 64 bit integers, making it possible to run tests with weights of considerable size. For technical reasons it was not possible to modify the FORTRAN codes MT2 and MThard to 64 bit integers.

The outcome of the experiments is shown in Tables 5.7 to 5.9. It is interesting to observe that uncorrelated and weakly correlated instances are almost unaffected by

the increased data range. However, for nearly all other instance types the problems become harder as the data range is increased. Several instances cannot be solved within the given time limit, and it is also interesting to note that the upper bounds used in Combo for some instances have difficulties in closing the gap between the upper and lower bound. The dynamic programming part of Minknap and Combo ensures that the optimal solution is found in pseudopolynomial time, but as the data range is increased this time bound starts to be unacceptably high.

The main observations from the computational experiments with small-range instances are however still valid. First of all, we notice that the methods used for finding tighter upper and lower bounds in Combo do pay off, since in general Combo is able to solve considerably more instances than the "clean" dynamic programming version Minknap. Next, we notice that branch-and-bound methods cannot compete with dynamic programming methods. There is actually a single exception to this observation for the subset sum instances. The randomly generated subset sum instances have numerous solution to the decision problem SSP-DECISION and hence a branch-and-bound algorithm may terminate the search as soon as an optimal solution has been found.

5.5.3 Difficult Instances With Small Coefficients

It is more challenging to construct instances with small coefficients, where present algorithms perform badly. Obviously, the worst-case complexity of dynamic programming algorithms still hold, but one may construct the instances so that it is difficult to fathom states through upper bound tests when using DP-with-Upper-Bounds. The following classes of instances have been identified as having the desired property.

- **spanner instances** span(v,m)
  These instances are constructed such that all items are multiples of a quite small set of items — the so-called spanner set. The spanner instances span(v,m) are characterized by the following three parameters: v is the size of the spanner set, m is the multiplier limit, and finally we may have any distribution (uncorrelated, weakly correlated, strongly correlated, etc.) of the items in the spanner set.

  More formally, the instances are generated as follows: A set of v items is generated with weights in the interval $[1,R]$, and profits according to the distribution. The items (p_k,w_k) in the spanner set are normalized by dividing the profits and weights with $m+1$. The n items are then constructed, by repeatedly choosing an item (p_k,w_k) from the spanner set, and a multiplier a randomly generated in the interval $[1,m]$. The new item has profit and weight $(a \cdot p_k, a \cdot w_k)$.

  Three distributions of the spanner problems span(s,m) have been considered: uncorrelated, weakly correlated, and strongly correlated. The multiplier limit was chosen as $m = 10$. Computational experiments showed that the instances became

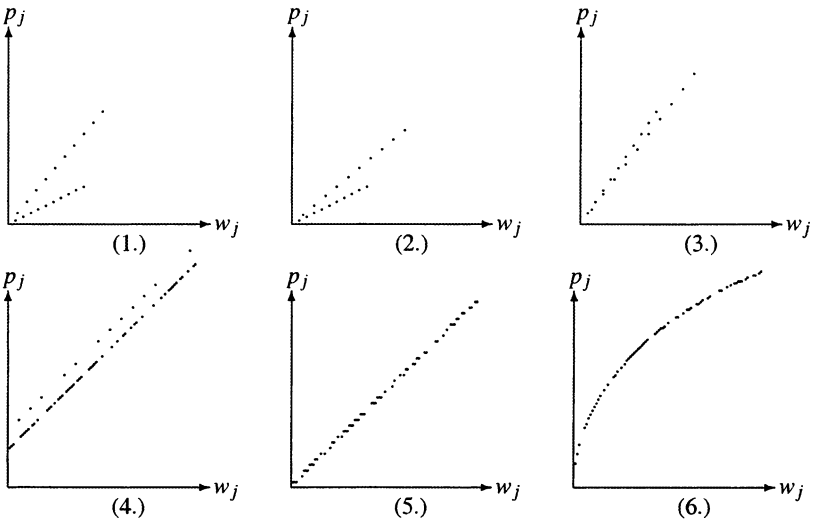


Fig. 5.15. New test instances considered. (1.) Uncorrelated spanner instances span(2,10). (2.) Weakly correlated spanner instances span(2,10). (3.) Strongly correlated spanner instances span(2,10). (4.) Multiple strongly correlated instances mstr($3R/10, 2R/10, 6$). (5.) Profit ceiling instances pceil(3). (6.) Circle instances circle($\frac{2}{3}$).

harder to solve for smaller spanner sets. Hence in the following we will consider the instances: uncorrelated span(2,10), weakly correlated span(2,10), and strongly correlated span(2,10).

- **multiple strongly correlated instances** mstr(k_1, k_2, d)
  These instances are constructed as a combination of two sets of strongly correlated instances. Both instances have profits $p_j := w_j + k_i$ where k_i, $i = 1,2$ is different for the two instances.

  The multiple strongly correlated instances mstr(k_1, k_2, d) are generated as follows: The weights of the n items are randomly distributed in $[1, R]$. If the weight w_j is divisible by d, then we set the profit $p_j := w_j + k_1$ otherwise set it to $p_j := w_j + k_2$. Notice that the weights w_j in the first group (i.e. where $p_j = w_j + k_1$) will all be multiples of d, so that using only these weights we can at most use $d\lfloor c/d \rfloor$ of the capacity. To obtain a completely filled knapsack we need to include some of the items from the second distribution.

  Computational experiments showed that very difficult instances could be obtained with the parameters mstr($3R/10, 2R/10, d$). Choosing $d = 6$ results in the most difficult instances, but values of d between 3 and 10 can all be used.

- **profit ceiling instances** pceil(d)
  These instances have the property that all profits are multiples of a given parameter d. The weights of the n items are randomly distributed in $[1, R]$, and the profits are set to $p_j = d\lceil w_j/d \rceil$.

n	uncorr. span(2,10)	weak. corr. span(2,10)	str. corr. span(2,10)	mstr(2,6)	pceil(3)	circle($\frac{2}{3}$)
20	0.1	0.6	0.5	0.0	0.2	0.1
50	9982.0	—	—	0.7	14.1	0.3
100	—	—	—	77.0	—	18.4
200	—	—	—	—	—	—
500	—	—	—	—	—	—
1000	—	—	—	—	—	—
2000	—	—	—	—	—	—
5000	—	—	—	—	—	—
10000	—	—	—	—	—	—

Table 5.10. Solution times in milliseconds, MT2, for difficult instances with small coefficients (INTEL PENTIUM III, 933 MHz).

n	uncorr. span(2,10)	weak. corr. span(2,10)	str. corr. span(2,10)	mstr(2,6)	pceil(3)	circle($\frac{2}{3}$)
20	0.1	0.4	0.3	0.0	0.2	0.0
50	2563.7	—	—	1.3	18.7	0.2
100	—	—	—	68.5	—	50.0
200	—	—	—	—	—	29522.6
500	—	—	—	—	—	—
1000	—	—	—	—	—	—
2000	—	—	—	—	—	—
5000	—	—	—	—	—	—
10000	—	—	—	—	—	—

Table 5.11. Solution times in milliseconds, Expknap, for difficult instances with small coefficients (INTEL PENTIUM III, 933 MHz).

The parameter d was experimentally chosen as $d = 3$, as this resulted in sufficiently difficult instances.

- **circle instances** circle(d)
  The instances circle(d) are generated such that the profits as function of the weight form an arc of a circle (actually an ellipsis). The weights are uniformly distributed in $[1,R]$ and for each weight w the corresponding profit is chosen as $p = d\sqrt{4R^2 - (w-2R)^2}$.

Since all the above "difficult" instances are generated with moderate data range, all the codes MT2, Expknap, Minknap, MThard and Combo could be used in the normal version using 32 bit integers. The outcome of the experiments is shown in Tables 5.10 to 5.14.

It appears, that the branch-and-bound algorithms MT2 and Expknap are able to solve this kind of instances only for small values of n. The MThard algorithm, which is a combination of branch-and-bound and dynamic programming has a slightly better performance, being able to solve instances of moderate size. Both of the two dynamic programming algorithms Minknap and Combo are able to solve also large-sized instances since the worst-case running time of $O(nc)$ is acceptable for small

n	uncorr. span(2,10)	weak. corr. span(2,10)	str. corr. span(2,10)	mstr(2,6)	pceil(3)	circle($\frac{2}{3}$)
20	0.0	0.0	0.0	0.0	0.1	0.0
50	0.0	0.2	0.3	0.0	0.3	0.3
100	0.3	0.5	0.4	0.9	0.9	1.0
200	0.8	1.8	1.7	2.5	2.7	2.7
500	4.2	8.2	7.0	8.7	18.2	9.5
1000	17.0	30.4	25.9	17.8	48.9	20.3
2000	67.6	136.7	108.7	37.5	179.7	43.2
5000	377.9	826.9	671.7	134.7	1368.7	152.1
10000	1695.1	3036.8	2579.0	332.6	5120.9	262.9

Table 5.12. Solution times in milliseconds, Minknap, for difficult instances with small coefficients (INTEL PENTIUM III, 933 MHz).

n	uncorr. span(2,10)	weak. corr. span(2,10)	str. corr. span(2,10)	mstr(2,6)	pceil(3)	circle($\frac{2}{3}$)
20	0.1	0.5	0.4	0.1	0.2	0.0
50	4.4	5.8	6.1	0.4	2.2	0.5
100	14.9	29.9	30.4	1.6	15.7	2.9
200	4618.6	6464.3	12986.1	11.6	28904.1	15.4
500	—	—	—	10945.0	—	79.0
1000	—	—	—	—	—	432.7
2000	—	—	—	—	—	—
5000	—	—	—	—	—	—
10000	—	—	—	—	—	—

Table 5.13. Solution times in milliseconds, MThard, for difficult instances with small coefficients.

n	uncorr. span(2,10)	weak. corr. span(2,10)	str. corr. span(2,10)	mstr(2,6)	pceil(3)	circle($\frac{2}{3}$)
20	0.0	0.0	0.0	0.0	0.2	0.0
50	0.1	0.1	0.0	0.1	0.4	0.0
100	0.2	0.5	0.5	0.5	0.8	0.5
200	0.7	1.6	1.6	1.3	2.3	1.1
500	3.6	7.6	7.8	4.2	14.9	4.6
1000	13.0	29.6	29.6	8.2	45.1	12.9
2000	50.6	136.8	147.8	20.3	149.7	31.0
5000	295.6	926.8	969.7	80.2	1216.2	130.0
10000	1319.1	3450.0	4186.0	202.1	4661.4	247.4

Table 5.14. Solution times in milliseconds, Combo, for difficult instances with small coefficients.

values of the data range. It is however interesting to observe, that the tighter bounds in Combo do not significantly improve the running times. Indeed, Minknap is able to solve some of the instances faster than Combo using only dynamic programming and cheap upper bounds U_0.

In conclusion we observe that none of the more advanced upper bounds U_4 and U_5 based on the addition of cardinality constraints, significantly contribute to the

solution of the present problems. Hence, research in upper bounds working on more general classes of problems should be stimulated.

6. Approximation Algorithms for the Knapsack Problem

Approximation algorithms and in particular approximation schemes like *PTAS* and *FPTAS* were already introduced in Section 2.5 and 2.6, respectively. The main motivation in these sections was to illustrate the basic concept of constructing simple approximation schemes. The focus was put on algorithms where both the correctness and the required complexities were easy to understand without having to go deeply into the details of complicated technical constructions. Hence, an intuitive understanding about the basic features of approximation should have been brought to the reader which is a necessary prerequisite to tackle the more sophisticated methods required to improve upon the performance of these simple algorithms.

In this chapter we will deal mainly with the best currently known approximation schemes and include only a brief review of earlier algorithms. Throughout this section the dependence on ε is usually reflected by $1/\varepsilon$. Since the algorithms work on integer numbers, we would theoretically have to round up $1/\varepsilon$ to the nearest integer. This will be frequently omitted for the sake of readability.

6.1 Polynomial Time Approximation Schemes

All the published *PTAS*s follow the same basic idea of "guessing" a certain set of items included in the optimal solution by going through all possible *candidate sets*, i.e. all subsets of items fulfilling some properties, and then filling the remaining capacity in some greedy way. All these algorithms do not require any relevant additional storage and hence can be performed within $O(n)$ space.

The classical *PTAS* by Sahni [415] requires $O(n^{\frac{1}{\varepsilon}})$ time. Already the simple *PTAS* defined by Algorithm H^{ε} in Section 2.6 had a better running time of $O(n^{\frac{1}{\varepsilon}-1})$. This was achieved by using the $1/2$–approximation ratio of Ext-Greedy in the analysis of the performance guarantee. Thereby, a reduction of the maximum cardinality ℓ of the guessed subset of items from $\frac{1}{\varepsilon}-1$ to $\frac{1}{\varepsilon}-2$ was made possible.

6.1.1 Improving the *PTAS* for (KP)

A further improvement of the running time by a factor of n compared to H^{ε}, i.e. n^2 compared to the *PTAS* in [415], was given in algorithm CKPP by Caprara, Kellerer,

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

Pferschy and Pisinger [65]. Their main point is *not* to run Ext-Greedy separately after choosing one of the $\binom{n}{\ell}$ subsets of "large profit values". Instead, they exploit a *monotonicity property* of the greedy algorithm to evaluate $O(n)$ subsets in one run. Monotonicity means that after sorting the items in decreasing order of their efficiencies, packing the items in this order yields a greedy solution not only for the single-capacity knapsack with capacity c but on the way also for the all-capacities version.

Recall that Greedy does not have to continue packing items after failing to pack an item for the first time in order to reach the $1/2$–approximation ratio. It was pointed out in Section 2.5 in connection with Theorem 2.5.4 that the simplified greedy algorithm Greedy-Split can be used instead of Greedy. Still the necessary comparison to the single item with largest profit as in Ext-Greedy must be taken into account.

In principle, algorithm *PTAS* proceeds in the same way as H^{ϵ}. However, to use the greedy algorithm for several subsets in one run we need to generate the subsets of cardinality ℓ in a more systematic way to enumerate sequences of subsets with decreasing capacity values. Assume that the items are sorted such that $p_1 \leq p_2 \leq \ldots \leq p_n$, and let any generated subset L with $|L| = \ell$ consist of items $\{j_1, j_2, \ldots, j_\ell\}$. Then the required property of the subset enumeration can be achieved by fixing at first the item j_1 with smallest profit and $j_{\ell-1}$, the one with second largest profit. For each such pair every possible subset T of $\ell - 3$ items with profit between p_{j_1} and $p_{j_{\ell-1}}$ is generated. Finally, for every choice of T the item with highest profit j_ℓ is systematically chosen by going through all items with profit at least $p_{j_{\ell-1}}$ in *decreasing* order of weights. Thus, the sets L resulting from the iteration over j_ℓ for a fixed choice of $j_1, \ldots, j_{\ell-1}$ also have decreasing weights.

Greedy-Split fills the increasing residual capacities of the knapsack with items from $S := \{j \in N \setminus L \mid p_j \leq p_{j_1}\}$ in decreasing order of efficiencies for all values of j_ℓ in linear time. Recall that the $1/2$–approximation ratio of Ext-Greedy is only attained if also the single item with largest profit is considered as a solution, however not the largest item from S, but the largest item which fits into the residual capacity $c - w(L)$. Therefore, a second iteration over j_ℓ in the reverse direction, namely in *increasing* order of weights, is performed. After sorting S in decreasing order of profits, we can compute again in linear time the required item by going through S. Whenever an item exceeds the (decreasing) residual capacity, we move on to the next item in S with smaller profit and continue to do so until an item is encountered which fits into the knapsack.

To avoid sorting procedures for every execution of Greedy-Split and for the computation of the largest fitting item we sort all items in S both by decreasing efficiencies $e_j = p_j/w_j$ and by decreasing profits. The items with profit at least $p_{j_{\ell-1}}$ (set R containing all candidates for j_ℓ) are sorted by decreasing weights after fixing j_1 resp. $j_{\ell-1}$. Note that set R remains unchanged for all choices of T since it depends only on the choice of the second largest item $j_{\ell-1}$.

A detailed formulation of the resulting algorithm CKPP following mostly the presentation in [65] (but with a more precise application of Greedy-Split) is given in

Algorithm CKPP:

$\ell := \min\{\lceil \frac{1}{\varepsilon} \rceil - 2, n\}$
renumber the items such that $p_1 \leq p_2 \leq \ldots \leq p_n$

generate all subsets L with cardinality less than ℓ
for all subsets $L \subset N$ with $|L| \leq \ell - 1$ do
    if $\sum_{j \in L} w_j \leq c$ then
        update the currently best solution

generate all subsets L with cardinality ℓ denoted by $\{j_1, j_2, \ldots, j_\ell\}$
for $j_1 := 1, \ldots, n - \ell + 1$ *smallest item in L*
    let $S := \{1, \ldots, j_1 - 1\}$ *small items for* Greedy-Split
    Store copies of S sorted in decreasing order of efficiencies
    and in decreasing order of profits.
    for $j_{\ell-1} := j_1 + \ell - 2, \ldots, n - 1$ *second largest item in L*
        let $R := \{j_{\ell-1} + 1, \ldots, n\}$ *candidates for j_ℓ*
        sort R in decreasing order of weights
        for every $T \subseteq \{j_1 + 1, \ldots, j_{\ell-1} - 1\}$ with $|T| = \ell - 3$
            consider all possible items j_ℓ with highest profit in L
            let j_ℓ be the item with highest weight in R
            Perform Greedy-Split on an instance of (KP) with
            item set S and capacity $c - W(L)$.
            update the currently best solution
            for $j_\ell \in R$ in *decreasing* order of weights
                Recompute Greedy-Split for the new capacity
                $c - W(L)$ starting from the previous solution.
                update the currently best solution
            end for j_ℓ
            for $j_\ell \in R$ in *increasing* order of weights
                Find the item in S with largest profit and weight
                at most $c - W(L)$.
                update the currently best solution
            end for j_ℓ
        end for T
    end for $j_{\ell-1}$
end for j_1

Fig. 6.1. Improved *PTAS* for (KP) avoiding the full execution of a greedy algorithm for every candidate set of items.

Figure 6.1. Note that the changes from H^ε concern in principle only the details of the loop over all subsets of cardinality ℓ.

The correctness of CKPP is obvious since all subsets of cardinality ℓ are enumerated and a greedy algorithm with performance guarantee $1/2$ is applied to the set of remaining items respectively to the remaining free capacity. Therefore, the proof of Theorem 2.6.2 can be applied without any changes. The improved running time is stated in the next theorem.

Theorem 6.1.1 *The running time of Algorithm* CKPP *is* $O(n^{\lceil \frac{1}{\varepsilon} \rceil - 2})$ *for* $\varepsilon < 1/3$.

Proof. The number of subsets with cardinality at most $\ell - 1$ can be very roughly bounded by $n^{\ell-1}$. A clever implementation would be able to evaluate each of them in constant time but even going through the items of every subset separately would yield a total running time of $O(n^{\lceil \frac{1}{\varepsilon} \rceil - 2})$ for this phase of algorithm CKPP.

The second phase, where all subsets of cardinality ℓ are considered, basically consists of two nested for–loops for j_1 and $j_{\ell-1}$. For all $O(n^2)$ executions of the inner loop over $j_{\ell-1}$ a set of $O(n)$ items is sorted and all subsets T with cardinality $\ell - 3$ are generated. Hence, there will be a total of $O(n^2\, n^{\ell-3})$, i.e. $O(n^{\ell-1})$, different sets T computed during the execution of the algorithm.

For each subset T we let j_ℓ go through all indices from $j_{\ell-1} + 1$ to n once in decreasing and once in increasing order of weight with the help of the sorted set R. Now we can compute the solution of Greedy-Split consecutively for all values of j_ℓ by inserting the small items in S sorted by decreasing efficiencies into a knapsack of capacity $c - w(L)$. As soon as the capacity is filled, we check for a possible update of the currently best solution and then go to the next j_ℓ which yields a capacity $c - w(L)$ not smaller than before. Therefore, we can continue the computation of Greedy-Split from the point it was stopped for the previous set L. Since this procedure goes through set S only once for all considered values of j_ℓ, the computation for every subset T takes only $O(n)$ time.

The greedy extension, i.e. the consideration of the best single item from S, is done in the second iteration over j_ℓ where the same sets L are generated as before but in the reverse order. Since $c - W(L)$ is decreasing in this case, a single scan through S sorted by decreasing profits is sufficient for finding the appropriate items (which are decreasing in weight) for every subset T.

Altogether, the total running time of this phase is $O(n\,(n^{\ell-1} + n \log n))$ which proves the claim of the theorem for $\ell \geq 3$.

For $\ell = 2$ the sorting of the sets S and R would dominate the effort of enumerating the subsets T. To ensure the total running time of $O(n^2)$ two auxiliary arrays are introduced. One array E contains all items in decreasing order of efficiencies, the second array W consists of all items in decreasing order of weights. After computing these arrays at the beginning of algorithm CKPP in $O(n \log n)$ time, the sets S resp. R can be generated in linear time for every j_1 by picking the items during one pass through E resp. W. □

The special case of $\ell = 1$, i.e. $1/3 \leq \varepsilon < 1/2$ must be treated separately. We will briefly describe an algorithm with $O(n \log n)$ running time since we are not aware of any published algorithm with matching performance.

Lemma 6.1.2 *There is a 2/3–approximation algorithm for (KP) with $O(n \log n)$ running time.*

Proof. We will only give an outline of the algorithm. More details about the required data representation can be found e.g. in [256]. As auxiliary structures we use an

array P containing all items in decreasing order of profits and (as above) an array E containing all items in decreasing order of efficiencies. Furthermore, we construct a binary tree "on top of" E in the following "bottom up" way. The entries of E induce leafs of the tree. Each leaf contains the weight of the corresponding item. Following the ordering of E the leafs are partitioned into pairs by combining neighbouring leafs. Each pair gives rise to a parent node containing as weight number the sum of its two generating leafs. Thus a new level of $n/2$ inner nodes of the tree is constructed. This procedure of combining two neighbouring nodes of the tree and putting their weight sum in the new parent node at a higher level is continued until only a single root node containing the weight sum of all items remains. If there is an odd number of nodes in a level then the last remaining node is simply elevated to the following level. Clearly, all these auxiliary structures can be built in $O(n \log n)$ time.

The tree structure can be used to find the greedy solution of a knapsack problem with arbitrary capacity $c' \leq c$ in $O(\log n)$ time by exploring nodes in the following way. The first explored node is the root. If the weight of the root is not larger than c' then all items can be packed into the knapsack and we are done. Otherwise, if the weight w of the "left" child node of the root (corresponding to the $n/2$ items with higher efficiency) is not larger than c' we can add all items corresponding to the subtree rooted at this node to the knapsack and continue the exploration at the "right" child node with the new remaining capacity $c' - w$. If w is larger than c', the exploration is continued at the "left" child with the original capacity c'. This procedure is repeated recursively down through the tree until we end up in a leaf node.

The algorithm utilizes this data structure as follows. We go through all "guesses" of j_1, i.e. the item with highest profit in the optimal solution, in decreasing order of profits. For each guess the remaining capacity is filled in a greedy way by the above tree structure. Since we want to use only items with profit smaller than the current guess we have to remove permanently from the tree every candidate for j_1 before computing the corresponding greedy solution. This can be done by deleting the leaf containing j_1 and going up to the root while subtracting the weight of j_1 from every node on this path. Note that no rebalancing operations are necessary since the height of the tree always remains in $O(\log n)$.

The operations for every guess require $O(\log n)$ time which yields the overall running time of $O(n \log n)$. The approximation guarantee carries over immediately by applying the proof of Theorem 2.6.2 with $\ell = 1$. □

The relevance of the improvement stated in Theorem 6.1.1 might seem to be a minor one, since the exponent of the running time is still unpleasantly large. However, recall that a polynomial time approximation scheme will only be applied for quite moderate values of ε. To give a better picture of the gain achieved by the above algorithms Table 6.1 lists the performance guarantee which can be reached by the old approach in [415] and the recent algorithm CKPP within the same running time

complexity. The value for $O(n)$ follows from Theorem 2.5.4 and Lemma 3.1.1. The approximation ratio reachable in $O(n \log n)$ time was shown above in Lemma 6.1.2.

time complexity	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(n^4)$	...	$O(n^\ell)$
Sahni [415]	1/2	1/2	1/2	2/3	3/4	...	$\frac{\ell-1}{\ell}$
CKPP [65] and Lemma 6.1.2	1/2	2/3	3/4	4/5	5/6	...	$\frac{\ell+1}{\ell+2}$

Table 6.1. Comparison of guaranteed approximation ratios of *PTAS* for (KP).

6.2 Fully Polynomial Time Approximation Schemes

All fully polynomial time approximation schemes for (KP) follow the same basic approach of scaling the profit space thus reducing the number of different profit values and applying dynamic programming by profits to the resulting instance.

However, the details both of generating the scaled problem instance and of solving it by a dynamic programming algorithm offer many opportunities for improvements. Since these improvements usually involve tedious technicalities and notation, we will start this section with a straightforward algorithm Basic-FPTAS following the description of an *FPTAS* in Section 2.6 before we move on to the more complicated algorithms. The *FPTAS* with the best complexity currently known, both for running time and space, was presented recently by Kellerer and Pferschy [267, 266]. Its description will form the main part of this section.

To get a first guess about the value of the optimal solution we run Ext-Greedy which computes a lower bound z^ℓ satisfying $z^\ell \leq z^* \leq 2z^\ell$ according to Theorem 2.5.4. In Basic-FPTAS the profit range is partitioned into intervals of equal range $z^\ell \varepsilon / n$. The scaling of profits can be done by setting all profit values in the same interval equal to the lower interval bound. A scaled profit index sp_j is used to denote the number of the corresponding interval. Thus, all scaled profits are multiples of $z^\ell \varepsilon / n$.

The remainder of the *FPTAS* consists only of performing DP-Profits (see Section 2.3) on the scaled profit space. This means that $y(q) = w$ signifies the existence of a subset of items with total profit equal to $q \cdot z^\ell \varepsilon / n$ and weight sum w. An algorithmic description of the resulting Basic-FPTAS is given in Figure 6.2.

Exactly the same arguments used for showing Theorem 2.6.6 also prove that Basic-FPTAS is indeed an *FPTAS* for (KP). Note that condition (2.19) from Section 2.6 is valid also for the scaling factor $z^\ell \varepsilon / n$ since $p_{\max} \leq z^\ell$. The running time of Basic-FPTAS is given by $O(n^2/\varepsilon)$ (consider the two nested for-loops over j and q). As before, the trivial space requirements of $O(n^2/\varepsilon)$ can be reduced to $O(n/\varepsilon)$ by the storage reduction scheme of Section 3.3.

Algorithm Basic-FPTAS:

compute z^ℓ with $2z^\ell \geq z^*$
partition N into n/ε subsets N_i with $i = 0, \ldots, n/\varepsilon - 1$
each N_i covers a range of $z^\ell \varepsilon/n$
set $N_i := \{j \mid iz^\ell\varepsilon/n < p_j \leq (i+1)z^\ell\varepsilon/n\}$
for $i := 0, \ldots, n/\varepsilon - 1$ do
  for all $j \in N_i$ do
    $p_j := iz^\ell\varepsilon/n$
    $sp_j := i$ *scaled profit index*
$y(0) := 0$
for $q := 1$ to $2n/\varepsilon$ do
  $y(q) := c+1$ *initialization*
for $j := 1$ to n do
  for $q := 2n/\varepsilon$ down to sp_j do *item j may be packed*
    if $y(q - sp_j) + w_j < y(q)$ then
      $y(q) := y(q - sp_j) + w_j$
  end for q
end for j
$z^A := \max\{q \mid y(q) \leq c\} \cdot z^\ell\varepsilon/n$

Fig. 6.2. Simple *FPTAS* with scaling of profits and dynamic programming by profits based on DP-Profits.

The earliest *FPTAS* for (KP) (and also one of the first *FPTAS* in general) was given by Ibarra and Kim [241] in 1975. Two important approximation schemes with advanced treatment of items and algorithmic fine tuning were presented some years later. The classical paper by Lawler [295] gave a refined scaling resp. partitioning of the items and several other algorithmic improvements. A second paper by Magazine and Oguz [314] contains among other features a partitioning and recombination technique to reduce the space requirements of the dynamic programming procedure. Since the approach of Kellerer and Pferschy uses some of the main features of these papers but improves upon them in several aspects, we will not give the previous algorithms in detail but concentrate on this recent state of the art algorithm FPKP from [267, 266].

author	running time	space
Basic-FPTAS	$O(n^2 \cdot 1/\varepsilon)$	$O(n^2 \cdot 1/\varepsilon)$
Ibarra, Kim [241]	$O(n\log n + 1/\varepsilon^2 \cdot \min\{1/\varepsilon^2 \log(1/\varepsilon), n\})$	$O(n + 1/\varepsilon^3)$
Lawler [295]	$O(n\log(1/\varepsilon) + 1/\varepsilon^4)$	$O(n + 1/\varepsilon^3)$
Magazine, Oguz [314]	$O(n^2 \log n \cdot 1/\varepsilon)$	$O(n \cdot 1/\varepsilon)$
Kellerer, Pferschy [267, 266]	$O(n \min\{\log n, \log(1/\varepsilon)\} + 1/\varepsilon^2 \log(1/\varepsilon) \cdot \min\{n, 1/\varepsilon \log(1/\varepsilon)\})$	$O(n + 1/\varepsilon^2)$

Table 6.2. Complexities of fully polynomial approximation schemes for (KP).

A comparison of all mentioned algorithms is given in Table 6.2. Note that for higher values of $1/\varepsilon$ the space requirement is usually considered to be a more serious bottleneck for practical applications than the running time. A statement by Lawler [295, p. 344] says that "bounds are intended to emphasize asymptotic behaviour in n, rather than ε" indicating that n is considered to be of much larger magnitude than $1/\varepsilon$. For the solution of a knapsack problem with a moderate number of items and a very high accuracy exact solution methods could be successfully applied. Under the resulting assumption $n \geq 1/\varepsilon$, the new method by Kellerer and Pferschy is superior to all previous approaches both in time and space.

It should also be mentioned that Liu [302] developed an *FPTAS* for (KP) based on the classical approximation scheme by Ibarra and Kim [241]. The time complexity of the algorithm in [302] is given by $O(n \min\{\log n, \log(1/\varepsilon)\} + 1/\varepsilon^{2+2\delta})$ where δ is of order $O(\log c/(1+\log c))$. The space requirements are $O(n + 1/\varepsilon^{2+\delta})$. Unfortunately, these bounds depend not only on n and ε, but also on c. Note that for large c, δ will be close to 1. Since both the space bound and the time complexity are dominated by FPKP (for extremely small values of c the time bounds may be close to each other), we will not go into any details of this *FPTAS*.

Since the algorithm we are going to describe is fairly complicated and many of its features require a thorough technical description we will split the presentation in several parts. At first we will deal with the scaling and reduction of the item set. Then a more general *vector merging problem* will be introduced, which can be used as a formulation of the dynamic programming procedure on the modified instance. After putting these modules together the item set of the approximate solution can be reconstructed by applying the recursive scheme from Section 3.3. However, we will give a full description of the resulting procedure adapted for the approximation of (KP) and including some useful improvements.

Summarizing the main new ideas of the algorithm from [267, 266], the improvement of space is attained by storing only one item for each dynamic programming entry instead of a complete subset of items. The required solution set is retrieved by a fine tuned version of the storage reduction technique of Section 3.3.

The time improvement is derived on one hand by a more involved partitioning and reduction of the profit space of the items. On the other hand this reduced profit structure is exploited in a refined dynamic programming strategy. In particular, the scaled problem instances consist only of a limited number of different profits but several items with different weights for each profit value. These instances (items with different weights but identical profits) can be seen as lying between (KP) (different profits, different weights) and (BKP) where item copies have identical profits and weights. The property of (BKP) was exploited for an efficient dynamic programming algorithm (see Section 7.2.2) but also the current weaker property can be used to improve the performance.

6.2.1 Scaling and Reduction of the Item Set

The basic approach to handle the item set is the following. In the beginning all items with "small" profits are separated and set aside. The range of all remaining "large" profit values is partitioned into intervals of identical length. Every such interval is further partitioned into subintervals of a length increasing with the profit value, which means that the higher the profit in an interval, the fewer subintervals are generated.

Then from each subinterval a certain number of items with smallest weights is selected and all profits of these items are decreased to the lower subinterval bound. The number of selected items of every subinterval decreases with increasing profit value of the current interval. A precise description of the resulting procedure Scaling-Reduction for the modification of the item set is given in Figure 6.3.

Before this partitioning into intervals a lower bound z^ℓ on the optimal solution value with

$$z^\ell \leq z^* \leq 2z^\ell \tag{6.1}$$

has to be computed, e.g. by Ext-Greedy. Furthermore, for technical reasons the accuracy is refined by setting

$$\varepsilon := \frac{1}{\lceil \frac{2}{\varepsilon} \rceil} \leq \frac{\varepsilon}{2}. \tag{6.2}$$

Note that after this modification $1/\varepsilon$ and $1/\varepsilon^2$ are integers. It will be exploited later on in the dynamic programming procedure that after running Scaling-Reduction all profit values of the remaining relevant items are multiples of $z^\ell\varepsilon^2$.

For the construction of an ε–approximation scheme it was shown by Kellerer and Pferschy [267] that the instance of (KP) attained by the application of Scaling-Reduction induces only a suitably bounded change of the optimal solution value. This will be done for the set of large items in the following Lemma 6.2.1. For a capacity $c' \leq c$ let z_L and $z_{\mathcal{L}}$ be the optimal solution values of a knapsack problem with item set L and item set $\mathcal{L}$, respectively, as defined in Figure 6.3.

Lemma 6.2.1

$$z_L \geq (1-\varepsilon)z_{\mathcal{L}}$$

Proof. Starting with the possible removal of items from every subset L_i^k we note that property (6.1) yields

$$\left\lceil \frac{2}{i\varepsilon} \right\rceil iz^\ell\varepsilon \geq 2z^\ell \geq z_{\mathcal{L}}.$$

Hence there can be at most $\lceil \frac{2}{i\varepsilon} \rceil$ items from L_i in any feasible solution. The possible reduction of each set L_i^k to $\lceil \frac{2}{i\varepsilon} \rceil$ items with minimal weight has no effect on the

Scaling-Reduction:

compute z^ℓ and modify ε
let $S := \{j \mid p_j \leq z^\ell \varepsilon\}$ be the set of *small items*
let $\mathcal{L} := \{j \mid p_j > z^\ell \varepsilon\}$ be the set of *large items*
partition $\mathcal{L}$ into $1/\varepsilon - 1$ subsets L_i with $i = 1, \ldots, 1/\varepsilon - 1$
$L_i := \{j \mid iz^\ell \varepsilon < p_j \leq (i+1)z^\ell \varepsilon\}$
each L_i covers a range of $z^\ell \varepsilon$
for $i := 1, \ldots, 1/\varepsilon - 1$ do
  partition L_i into $\lceil \frac{1}{i\varepsilon} \rceil - 1$ subintervals L_i^k of range $iz^\ell \varepsilon^2$
  with $k = 1, \ldots, \lceil \frac{1}{i\varepsilon} \rceil - 1$ such that
  $L_i^k := \{j \mid iz^\ell \varepsilon(1 + (k-1)\varepsilon) < p_j \leq iz^\ell \varepsilon(1 + k\varepsilon)\}$
  and the remaining (possibly smaller) subinterval
  $L_i^{\lceil \frac{1}{i\varepsilon} \rceil} := \{j \mid iz^\ell \varepsilon(1 + (\lceil \frac{1}{i\varepsilon} \rceil - 1)\varepsilon) < p_j \leq (i+1)z^\ell \varepsilon\}$
  for $k := 1, \ldots, \lceil \frac{1}{i\varepsilon} \rceil$ do
    for all $j \in L_i^k$ do $p_j := iz^\ell \varepsilon(1 + (k-1)\varepsilon)$
    reduce profits to lower interval bound
    if $(|L_i^k| > \lceil \frac{2}{i\varepsilon} \rceil)$ then
      reduce L_i^k to the $\lceil \frac{2}{i\varepsilon} \rceil$ items with *minimal weight*
      delete all other items in L_i^k
  end for k
end for i
denote by $L \subseteq \mathcal{L}$ the set of all remaining large items

Fig. 6.3. Partitioning, scaling and reduction of the item set N yielding a set of small items S and a reduced set L of large items with scaled profit values.

total value of the corresponding optimal solution because selecting from items with identical profits those with smallest weight will not decrease the optimal solution value.

Concerning the effect of the reduction of profit values, we let μ_i be the number of items from L_i in an optimal solution of the instance with item set $\mathcal{L}$. As each item of L_i has a profit greater than $iz^\ell \varepsilon$ we get immediately

$$z_{\mathcal{L}} > \sum_{i=1}^{1/\varepsilon - 1} \mu_i i z^\ell \varepsilon. \tag{6.3}$$

However, the rounding down of the profits performed in the scaling procedure through the assignment $p_j := iz^\ell \varepsilon(1 + (k-1)\varepsilon)$ for $j \in L_i^k$ diminishes the profit of every item in L_i by at most $iz^\ell \varepsilon^2$. Bounding the modified total profit of the optimal item set as above we get

$$z_L \geq z_{\mathcal{L}} - \sum_{i=1}^{1/\varepsilon - 1} \mu_i i z^\ell \varepsilon^2 \geq z_{\mathcal{L}} - \varepsilon z_{\mathcal{L}} \geq (1 - \varepsilon) z_{\mathcal{L}}$$

by inserting (6.3). □

Since all items in one subinterval L_i^k are assigned the same profit value, it will be interesting to consider the *number of different profit values* $D(L)$ for an item set L.

Lemma 6.2.2 *After performing* Scaling-Reduction, $|L|$ *is* $O(\min\{n, 1/\varepsilon^2\})$ *and* $D(L)$ *is* $O(\min\{n, 1/\varepsilon \log(1/\varepsilon)\})$.

Proof. The maximal number of items in L is bounded by

$$\sum_{i=1}^{1/\varepsilon-1} \left\lceil \frac{1}{i\varepsilon} \right\rceil \cdot \left\lceil \frac{2}{i\varepsilon} \right\rceil \approx \frac{2}{\varepsilon^2} \sum_{i=1}^{1/\varepsilon} \frac{1}{i^2}, \tag{6.4}$$

which is of order $O(1/\varepsilon^2)$. The number of distinct profit values $D(L)$ in the set of large items L is bounded by

$$D(L) \leq \sum_{i=1}^{1/\varepsilon-1} \left\lceil \frac{1}{i\varepsilon} \right\rceil \approx \frac{1}{\varepsilon} \sum_{i=1}^{1/\varepsilon} \frac{1}{i} \approx \frac{1}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right) \tag{6.5}$$

and hence is in $O(1/\varepsilon \log(1/\varepsilon))$. The trivial bound of n, which is the total number of items, leads to the minimum expressions. □

It should be noted that a scaling procedure applying a closely related interval partitioning into equal sized and geometrically increasing ranges was recently proposed by Mastrolilli and Hutter [337]. Their combination of arithmetic and geometric rounding was applied successfully to the cardinality constrained knapsack problem and will be discussed in Section 9.7.4.

6.2.2 An Auxiliary Vector Merging Problem

After scaling the profit space of a (KP) instance, dynamic programming by profits as introduced in Section 2.3 is performed to compute an optimal solution of the modified instance. However, rather than applying a standard algorithm like DP-Profits it is worth to devote more care to the development of a dynamic programming scheme which takes the simplified profit structure of the scaled instance with many identical profit values into account.

As a generalization of the resulting task a vector merging problem was introduced by Kellerer and Pferschy [266]. Their solution method uses an advanced dynamic programming technique which does not perform single update operations one after the other but identifies intervals of array entries which are updated from the same "origin" position. Our presentation will follow closely the formulation given in [266] (omitting an intermediate version of the algorithm).

VectorMerge (VM):

Given two vectors $A = (A(1),\ldots,A(n))$ and $B = (B(0),\ldots,B(n-1))$ with $0 \le B(0) \le B(1) \le \ldots \le B(n-1)$ we would like to compute a vector C of the same size defined by

$$\begin{aligned}
C(1) &:= A(1)+B(0),\\
C(2) &:= \min\{A(1)+B(0)+B(1), A(2)+B(0)\},\\
C(3) &:= \min\{A(1)+B(0)+B(1)+B(2), A(2)+B(0)+B(1),\\
&\qquad A(3)+B(0)\},\\
&\vdots\\
C(n) &:= \min\{A(1)+B(0)+B(1)+\ldots+B(n-1),\ldots,\\
&\qquad A(n-1)+B(0)+B(1), A(n)+B(0)\}.
\end{aligned}$$

Equivalently, we can write

$$C(k) := \min\left\{ A(\ell) + \sum_{j=0}^{k-\ell} B(j) \;\middle|\; \ell = 1,\ldots,k \right\},\ k = 1,\ldots,n. \qquad (6.6)$$

The computation of vector C can be done trivially in $O(n^2)$ time, e.g. by evaluating the minimum over at most n values explicitly for each entry $C(k)$ and storing the required partial sums for the next iteration with $C(k+1)$. In the following we will develop an algorithm with an improved time complexity of $O(n \log n)$.

To provide a better intuition of the applied technique we will start with a brief description of a straightforward solution method still requiring $O(n^2)$ time but performing the computation in a different order. In a first iteration we check for all entries of C whether $A(1)$ plus the appropriate entries of B yield new minima, then we do the same for $A(2)$ and so on. Basically, the values in the above definition are computed columnwise instead of rowwise.

For notational convenience we introduce an n-dimensional vector *origin* indicating finally that the minimum defining $C(k)$ is attained by $A(origin(k))$ plus the appropriate entries of B. Note that C is completely determined by *origin*. Through this representation of C we can identify a monotonicity property in problem (VM) which will be exploited in the improved algorithm.

Lemma 6.2.3 *There exists a solution of (VM) with*

$$origin(j) \le origin(j+1) \text{ for } j = 1,\ldots,n-1.$$

Proof. Assume otherwise that in every solution there is some j with $i := origin(j) > origin(j+1) =: i'$. Then there would be

$$\begin{aligned} C(j+1) &= A(i') + B(0) + B(1) + \ldots + B(j+1-i') \\ &\leq A(i) + B(0) + B(1) + \ldots + B(j+1-i), \end{aligned}$$

by definition of i'. Considering that $B(j+1-i') \geq B(j+1-i)$ the inequality holds even more so after subtracting the last term from both sums. This yields

$$A(i') + B(0) + B(1) + \ldots + B(j-i') \leq A(i) + B(0) + B(1) + \ldots + B(j-i) = C(j),$$

which is either a contradiction to the definition of C, or (in the case of equality) there also exists a solution where this violation of the desired property can be avoided by a different breaking of ties in the minimum. □

The following approach will give a solution fulfilling the property of Lemma 6.2.3. In the application of (VM) to the *FPTAS* for (KP), but very likely also in other applications of this dynamic programming structure, it will turn out that the vector *origin* contains identical values for a large number of consecutive entries. This gives rise to the idea not to store and compute each of the values of *origin* explicitly but to try to find only the endpoints of intervals in which *origin* remains constant. This means that for each i we define the endpoints $a(i), b(i)$ of an interval such that for every j with $a(i) \leq j \leq b(i)$ there is $origin(j) = i$ and hence $C(j) = A(i) + B(0) + B(1) + \ldots + B(j-i)$.

Clearly, some (more likely many) values of i will not appear in *origin* at all and hence will be assigned an empty interval with $a(i) := b(i) := \infty$. To achieve an efficient handling of the non-empty intervals they will be linked together by defining a vector *pred* where an entry $pred(i) = \ell$ means that $a(i), b(i) \neq \infty$, $a(k) = b(k) = \infty$ for $\ell < k < i$ and $a(\ell), b(\ell) \neq \infty$. For $a(i) \neq \infty$ there always must be $a(i) = b(pred(i)) + 1$.

The solution of (VM) by a computation of these intervals still requires $O(n^2)$ running time. However, we can improve the computation in the following way. Note that if $C(j) > A(i) + B(0) + B(1) + \ldots + B(j-i)$ holds for $j = a(k)$ for some k, it follows from Lemma 6.2.3 that this inequality also holds for all $j > a(k)$. Hence, it is sufficient to check this condition for all starting points $a(k)$ of non-empty intervals. As soon as we find a position $a(k)$ where the condition is violated we have to find the exact new values of $b(k)$ and $a(i)$. This can be done by binary search between $a(k)$ and the old value of $b(k)$. The resulting algorithm VectorMerge-Interval is presented in Figure 6.4. Its running time was analyzed by Kellerer and Pferschy [266] and is given in the following theorem.

Theorem 6.2.4 *Algorithm* VectorMerge-Interval *runs in* $O(n \log n)$ *time.*

Proof. We consider at first the comparisons of $C(j)$ in the while-loop for some previously detected interval endpoints. If the inequality is true, then the corresponding interval is deleted and never considered again. If the inequality is false, then the loop

Algorithm VectorMerge-Interval:

for $j := 1$ to n do
  $C(j) := A(1) + B(0) + B(1) + \ldots + B(j-1)$
$a(1) := 1$, $b(1) := n$, $pred(1) := 0$
$last := 1$ *index of the most recent non-empty interval*
for $i := 2$ to n do
  if $C(n) > A(i) + B(0) + B(1) + \ldots + B(n-i)$ then
    $b(i) := n$
    $j := a(last)$
    while $C(j) > A(i) + B(0) + \ldots + B(j-i)$ and $j \geq i$ do
      $C(j) := A(i) + B(0) + B(1) + \ldots + B(j-i)$
      $a(last) := \infty$, $b(last) := \infty$
      $last := pred(last)$
      $j := a(last)$
    end while

    if $j \geq i$ then
      perform binary search to find the largest value j in interval $a(last), \ldots, b(last)$ with
      $C(j) \leq A(i) + B(0) + B(1) + \ldots + B(j-i)$
      $b(last) := j$, $a(i) := j+1$
    else
      $b(last) := i-1$, $a(i) := i$
    end if
    $pred(i) := last$
    $last := i$
  else
    $a(i) := \infty$, $b(i) := \infty$
  end if
end for i

reconstruction of origin:
while $last > 0$ do
  for $j := b(last)$ down to $a(last)$ do
    $origin(j) := last$
  $last := pred(last)$
end while

Fig. 6.4. Interval based dynamic programming to solve problem (VM).

is stopped. Hence, the total number of these comparisons is linear in n because there can be at most n positive results (one for every interval) and at most one negative result in each of the n iterations of the for-loop.

Thus the running time of VectorMerge-Interval is dominated by not more than n executions of the binary search over $O(n)$ values. In order to perform the binary search procedure in $O(\log n)$ time, i.e. to answer every query of the binary search in constant time, we generate an n-dimensional auxiliary array where every entry j, $j = 0, \ldots, n-1$, contains the value $B(0) + B(1) + \ldots + B(j)$. Now the overall running time of $O(n \log n)$ follows. □

6.2.3 Solving the Reduced Problem

As indicated above, we will apply dynamic programming by profits as defined by recursion (2.10) to compute the solution of an instance with the simplified profit structure attained by Scaling-Reduction from Section 6.2.1. Note that we will compute only the array of weights for every profit value but not the corresponding item sets. However, we will record for every entry of the dynamic programming array $y(q)$ the *most recently added item* $r(q)$, i.e. the item causing the latest update. The technical details of recording these items should be fairly obvious and are omitted for the sake of clarity.

Recursion (2.10) can be done in a more efficient way for the modified item set attained by the scaling procedure.

Clearly, the computation of $y_j(q)$ can be done in an arbitrary order of the profit values q. It will be convenient to evaluate in one iteration those entries with the same residual value of the division of q by the profit p_t of the item currently added to the problem. This means that for every residual value $r = 0, \ldots, p_t - 1$ we consider the new entries $y_j(r), y_j(r+p_t), y_j(r+2p_t), \ldots$ in one run.

Furthermore, we will consider all items with identical profit p_t and weights $w_1^t \leq w_2^t \leq \ldots \leq w_m^t$ together in one iteration. Let the corresponding items be denoted by $j+1, j+2, \ldots, j+m$. For simplicity, we will illustrate the case $r = 0$. In the resulting iteration this means that e.g. instead of computing

$$\begin{aligned} y_j(p_t) &:= \min\{y_{j-1}(p_t), y_{j-1}(0) + w_1^t\}, \\ y_j(2p_t) &:= \min\{y_{j-1}(2p_t), y_{j-1}(p_t) + w_1^t\} \text{ and} \\ y_{j+1}(2p_t) &:= \min\{y_j(2p_t), y_j(p_t) + w_2^t\}, \end{aligned}$$

we determine

$$y_{j+1}(2p_t) := \min\{y_{j-1}(2p_t), y_{j-1}(p_t) + w_1^t, y_{j-1}(0) + w_1^t + w_2^t\}.$$

This formulation of the recursion has exactly the same structure as the vector merging problem (VM). Let $C(k) := y_{j-1+m}((k-1)p_t)$ and $A(k) := y_{j-1}((k-1)p_t)$ for $k = 1, \ldots, \lfloor 2z^\ell / p_t \rfloor + 1$. Vector B is chosen as $B(0) := 0$, $B(k) := w_k^t$ for $k = 1, \ldots, m$ and $B(k) := \infty$ for $k = m+1, \ldots, \lfloor 2z^\ell / p_t \rfloor$. The analogous treatment of the cases $r > 0$ is straightforward.

Thus, we have established a procedure Interval-Dynamic-Programming (L', q) described in detail in Figure 6.5. It computes for an instance of (KP) with scaled item set $L' \subseteq L$ an array y containing the minimal weight necessary to reach every profit value up to $q \leq 2z^\ell$ and an array r (not included in Figure 6.5) with the index of the item most recently added to the corresponding subset of items (i.e. the item with highest index). The procedure applies VectorMerge-Interval for every profit value p_t in L' and for every residual value $r = 0, \ldots, p_t - 1$ of a division by p_t. Recalling

```
Interval-Dynamic-Programming (L', q):
for k := 1, ..., ⌊q/(z^ℓ ε^2)⌋ do
    y(k(z^ℓ ε^2)) := c + 1      initialization, profits are multiples of z^ℓ ε^2
y(0) := 0
for all different profit values p_t ∈ L' do
    let w'_1 ≤ ... ≤ w'_m be the weights of the m items with profit p_t
    for r := 0, ..., p_t − 1 do      all residual values
        for k := 1, ..., ⌊q/p_t⌋ + 1 do
            A(k) := y((k − 1)p_t + r)
            if k ≤ m then B(k) := w'_k
            else B(k) := ∞
        B(0) := 0
        perform VectorMerge-Interval with A and B returning C
        for k := 1, ..., ⌊q/p_t⌋ + 1 do
            y((k − 1)p_t + r) := C(k)
    end for
end for
```

Fig. 6.5. Dynamic programming on the scaled instance invoking subproblems of type (VM).

that $D(L')$ denotes the number of different profit values of a set L' its performance is given as follows.

Lemma 6.2.5 Interval-Dynamic-Programming (L', q) *solves an instance of (KP) with an item set* $L' \subseteq L$ *and a total profit of at most* $q \leq 2z^\ell$ *in* $O(D(L') \cdot z^\ell \log(z^\ell))$ *time and* $O(|L'| + z^\ell)$ *space.*

Proof. Interval-Dynamic-Programming performs a total of $D(L')$ iterations over all different profit values p_t. In each iteration a total number of p_t subproblems (one for every residual value of a division by p_t) has to be solved. Each subproblem corresponds to an appropriate instance of (VM) with arrays of length at most $(2z^\ell)/p_t$. With Theorem 6.2.4 such an instance can be solved in $O(z^\ell/p_t \cdot \log(z^\ell))$ time. Summing up over all subproblems and all iterations yields the claimed running time.

Concerning the space requirements, it is obviously sufficient to store the item set L' and the dynamic programming arrays y and r both of length $2z^\ell$. The organization of such an array was discussed in Section 2.3. □

It is an interesting open problem to find a further improvement getting rid of the $\log(z^\ell)$ factor thus making the running time complexity equal to the product of the number of entries of the dynamic programming array and the number of different item profits.

6.2.4 Putting the Pieces Together

Combining the algorithms from Sections 6.2.1 and 6.2.3, namely performing the procedures Scaling-Reduction and Interval-Dynamic-Programming $(L, 2z^\ell)$, we can compute a $(1-\varepsilon)$-approximation of the optimal solution value of the instance consisting only of large items as implied by Lemma 6.2.1.

Taking also the small items into account it should be noted that these may play an important role in an optimal or approximate solution. It may well be the case that *all* solutions close to the optimum consist *only* of small items, e.g. if their efficiencies are much higher than the those of the large items. Therefore, we will check how much profit is contributed by the large and by the small items. This can be done by going through the dynamic programming array y for all profit values q, which are multiples of $z^\ell\varepsilon^2$, from 0 to $2z^\ell$ and adding small items to a knapsack with residual capacity $c-y(q)$ in a greedy-type way (of course only if $y(q) \leq c$). Naturally, out of the resulting $2z^\ell+1$ combined profit values, the highest one is selected and denoted by z^A.

Algorithm FPKP:

perform Scaling-Reduction
$z^A := 0$, $X^A := \emptyset$
$X_L := \emptyset$ *large items in the solution set*
$X_S := \emptyset$ *small items in the solution set*
perform Interval-Dynamic-Programming $(L, 2z^\ell)$
returning (y, r)

sort S in decreasing order of efficiencies
for all profits $q \leq 2z^\ell$ with $y(q) \leq c$ do in decreasing order of $y(q)$
add up items from S as long as their weight sum is not more than $c-y(q)$ yielding a profit of z_S
if $z^A < q+z_S$ then
$z^A := q+z_S$, $z_L := q$
let $X_S \subseteq S$ contain items with profit $z^A - z_L$ from iteration z_L

perform Backtracking (y, r, z_L) returning (z^N)
perform Recursion $(L, z_L - z^N)$
$X^A := X_L \cup X_S$

Fig. 6.6. Main part of the FPKP: The best possible combination of an approximate solution of large items with a greedy solution of small items is determined.

A formal description of the resulting algorithm FPKP by Kellerer and Pferschy [267, 266] can be found in Figure 6.6. It can be shown that FPKP yields a $(1-\varepsilon)$-approximation of the overall optimal solution value.

Lemma 6.2.6 *The solution z^A computed by* FPKP *satisfies*

$$z^A \geq (1-\varepsilon)z^*.$$

Proof. As usual we will compare the solution value resulting from the above algorithm with an optimal solution. Let us partition the optimal solution value into $z^* = z_L^* + z_S^*$, where z_L^* denotes the part contributed by large items and z_S^* the part summing up the small items in the optimal solution. The corresponding total weight of the large items will be denoted by $c_L^* \leq c$. In Lemma 6.2.1 it was shown that there exists also a set of items in L with total value at least $z_L^* - \varepsilon z_L^*$ and weight at most c_L^*.

Hence, it follows from Lemma 6.2.5 that procedure Interval-Dynamic-Programming $(L, 2z^\ell)$ generates an entry in the dynamic programming array corresponding to a set of items with profit between $(1-\varepsilon)z_L^*$ and z_L^* and weight not greater than c_L^*.

In the above algorithm at some point during the iteration over all profit values small items are added to the corresponding set of items in a greedy way to fill as much as possible of the remaining capacity which is at least $c - c_L^*$. It is known from Corollary 2.2.3 that the profit difference between an optimal algorithm and the greedy heuristic is less than the largest profit of an item and hence at most $z^\ell\varepsilon$. All together, this yields with Lemma 6.2.1

$$z^A \geq (1-\varepsilon)z_L^* + z_S^* - z^\ell\varepsilon = z^* - \varepsilon(z_L^* + z^\ell) \geq z^* - 2\varepsilon z^*$$

which proves the claim after modifying ε according to (6.2) in Section 6.2.1. □

The reconstruction of the item set $X_L \subseteq L$ leading to this profit value is more complicated. Clearly, it would be possible to store the complete subset of large items corresponding to every array entry of y. However, this would increase the memory requirement of the dynamic programming array by a factor of $O(1/\varepsilon^2)$.

A natural idea would be to exploit array r to reconstruct X_L. Clearly, the most recently added item $j := r(z_L)$ is part of this set. Inspecting $r(z_L - p_j)$ may yield a second item of X_L. Unfortunately, it is not possible to continue going back through the entries of r along this line and collect the complete set X_L. Note that the entries encountered in this sequence may have been updated *after* being (implicitly) used for an update of $y(z_L)$. In particular, the same item may have been added to several of them and thus an item may appear more than once in the resulting item set.

A different strategy solves this problem of finding X_L by applying the storage reduction scheme of Section 3.3. Note that Interval-Dynamic-Programming fulfills the necessary conditions of procedure Solve in Recursive-DP. To illustrate the more sophisticated application of this idea in the context of the current approximation scheme and for the sake of consistency we will give a full, but simplified description of the resulting procedure Recursion from [267] in Figure 6.7.

Recursion $(\tilde{L}, \tilde{q})$:

Divide
partition $\tilde{L}$ into two disjoint subsets $\tilde{L}_1, \tilde{L}_2$ with
$D(\tilde{L}_1) \approx D(\tilde{L}_2) \approx D(\tilde{L})/2$
perform Interval-Dynamic-Programming $(\tilde{L}_1, \tilde{q})$
returning (y_1, r_1)
perform Interval-Dynamic-Programming $(\tilde{L}_2, \tilde{q})$
returning (y_2, r_2)

Conquer
find indices q_1, q_2 such that
$q_1 + q_2 = \tilde{q}$ and $y_1(q_1) + y_2(q_2)$ is minimal
$(*)$ reorganize the storage structure

Recursion
perform Backtracking (y_1, r_1, q_1) returning (z_1^N)
if $(q_1 - z_1^N > 0)$ then
perform Recursion $(\tilde{L}_1, q_1 - z_1^N)$
perform Backtracking (y_2, r_2, q_2) returning (z_2^N)
if $(q_2 - z_2^N > 0)$ then
perform Recursion $(\tilde{L}_2, q_2 - z_2^N)$

Fig. 6.7. Recursive procedure to reconstruct a solution set $X_{\tilde{L}}$ with value $\tilde{q}$.

Backtracking (y, r, q):

$z^N := 0$ *collected part of the solution value*
repeat
$j := r(q)$ *item j is identified as part of the solution*
$X_L := X_L \cup \{j\}$
$q := q - p_j$
$z^N := z^N + p_j$
until $(q = 0$ or $y(q) < y(q + p_j) - w_j)$
return (z^N)

Fig. 6.8. Backtracking procedure to reconstruct a part of X_L from the dynamic programming array.

Compared to the procedure in Section 3.3 a major improvement can be attained by following the above idea and reconstructing a part of the solution set from every dynamic programming computation instead of generating the solution set only at the last level of the recursion.

This means that the entries of r are used to find items of X_L as far as possible before entering the next recursion step. In particular, we can "follow" the entries of r as

long as no entry is reached which was updated after X_L was first determined. This can be checked easily by comparing the actual values found in y with the values expected from following the sequence of items found so far. Hence, the actual solution set is collected during the successive executions of the corresponding procedure Backtracking in Figure 6.8 and not by Recursion itself.

To improve the practical performance of the algorithm it is recommended to keep track of the order in which items are considered in Interval-Dynamic-Programming, i.e. by sticking to increasing item numbers. If j' is the last item added to X_L during one execution of Backtracking, we can delete at the end of Backtracking all items which were considered in Interval-Dynamic-Programming after item j' because they were not needed to reach the solution value p.

The correctness of this recursive computation of X^A might be concluded from the general Theorem 3.3.1. However, since the recursive storage reduction is used in a quite specialized way for FPKP which differs considerably from the general scheme in Section 3.3, a self-contained proof seems to be called for.

Lemma 6.2.7 *If there exists a feasible subset of $\tilde{L}$ with total profit $\tilde{q}$, then the execution of* Recursion $(\tilde{L},\tilde{q})$ *will add such a subset to X_L.*

Proof. Naturally, a feasible subset of items $X \subseteq \tilde{L}$ with profit $\tilde{q}$ can be divided in **Conquer** into two parts, $X_1 \subseteq \tilde{L}_1$ with total profit q_1 and $X_2 \subseteq \tilde{L}_2$ with total profit q_2. By definition of the dynamic programming arrays, the entries $y_1(q_1)$ resp. $y_2(q_2)$, as computed by Interval-Dynamic-Programming, refer to the subsets X_1 resp. X_2. The effects of Backtracking are described by the following statement.

> **Claim:** After performing Backtracking (y,r,q) a subset of items with total profit z^N was added to X_L and there exists a subset of items with total profit $q-z^N$.

The correctness of this claim is based on the fact that Backtracking is always performed after Interval-Dynamic-Programming. By construction, the entry $y(q)$ is based on the existence of a subset of items with profit q. If the repeat loop is finished by the first stopping criterion the whole set of items with profit $z^N = q$ was added to X_L. Otherwise, we stop because an entry was updated after it was used in the sequence of entries leading to $y(q)$. However, there must have been a previous entry at this position which was part in this sequence and which represented a subset with profit $q-z^N$.

To analyze the overall effect of executing Recursion also the recursive calls for $\tilde{L}_1$ and $\tilde{L}_2$ must be analyzed. This recursive structure of Recursion corresponds to an ordered, binary rooted tree as elaborated in the discussion of Theorem 3.3.1. Each

node in the tree corresponds to a call to Recursion with the root corresponding to the first call in the main part of FPKP. A node may have up to two *child nodes*, the *left* child corresponding to the call of Recursion with $\tilde{L}_1$ and the *right* child corresponding to a call with $\tilde{L}_2$. This tree model will be used again in the proof of Theorem 6.2.10.

The statement of the Lemma will be shown by backwards induction moving "upwards" in the tree, i.e. starting with its leafs and applying induction to the inner nodes.

The leafs of the tree are executions of Recursion with no further recursive calls. Hence, the two corresponding conditions yield $z_1^N + z_2^N = q_1 + q_2 = \tilde{q}$ and the statement of the Lemma follows from the claim concerning Backtracking.

If the node under consideration has one or two children, the corresponding calls of Recursion follow immediately after Backtracking on the same set of parameters. The above claim guarantees that the existence condition required for the application of the induction hypothesis to the recursive execution of Recursion is fulfilled. By induction, during the processing of the child nodes items with total profit $q_1 - z_1^N + q_2 - z_2^N$ are added to X_L. Together with the items of profit z_1^N and z_2^N added by Backtracking this proves the above statement. □

Lemmata 6.2.6 and 6.2.7 can be summarized in

Theorem 6.2.8 *Algorithm* FPKP *is an* ε*–approximation scheme for* (KP).

Proof. Lemma 6.2.6 shows that the solution value of FPKP, which can be written as $z^A = z_L + z_S$, is close enough to the optimal solution value. During the main part of FPKP small items with profit z_S are collected in X_S. The first execution of Backtracking puts items with profit z^N into X_L. From the definition of dynamic programming we know the existence of an item set with profit $z_L - z^N$. Lemma 6.2.7 guarantees that the complete set X^A is finally determined. □

It remains to analyze the asymptotic running time and space requirements of FPKP. At first we will adapt Lemma 6.2.5 to scaled instances.

Corollary 6.2.9 *After* Scaling-Reduction *the resulting instance of* (KP) *with item set* $L' \subseteq L$ *and total profit at most* $q' \leq 2z^\ell$ *can be solved by* Interval-Dynamic-Programming (L', q') *in* $O(D(L') \cdot q'/(z^\ell \varepsilon^2) \log(q'/(z^\ell \varepsilon)))$ *time and* $O(|L'| + 1/\varepsilon^2)$ *space.*

Proof. After Scaling-Reduction all profit values of the large items are multiples of $z^\ell \varepsilon^2$. Interval-Dynamic-Programming performs only additions of profit values. Hence, we can easily divide all item profits by $z^\ell \varepsilon^2$ thus reducing by the same factor all profit sums and also the complexities given in Lemma 6.2.5. □

Now we can finally state the complexity of FPKP as derived by by Kellerer and Pferschy [267, 266].

Theorem 6.2.10 *For every performance ratio* $(1-\varepsilon)$, $0 < \varepsilon < 1$, *algorithm* FPKP *requires a running time of*

$$O(n \cdot \min\{\log n, \log(1/\varepsilon)\} + 1/\varepsilon^2 \log(1/\varepsilon) \cdot \min\{n, 1/\varepsilon \log(1/\varepsilon)\}) \tag{6.7}$$

and $O(n + 1/\varepsilon^2)$ *space.*

Proof. The selection of the items in every interval L_i^k by Scaling-Reduction can be done by a linear median algorithm. Considering also the number of intervals, Lemma 6.2.2 yields a time bound of $O(n + 1/\varepsilon \log(1/\varepsilon))$ for this first procedure.

The main computational effort is spent in the dynamic programming procedure. Since Backtracking is always performed on an array whose entries were previously computed by the dynamic programming procedure and since this array is scanned at most once, the running time of Backtracking is cleary dominated by the dynamic programming part.

It follows immediately from Corollary 6.2.9 and Lemma 6.2.2 that the first execution of Interval-Dynamic-Programming $(L, 2z^\ell)$ in the main part of the algorithm requires $O(\min\{n, 1/\varepsilon \log(1/\varepsilon)\} \cdot 1/\varepsilon^2 \log(1/\varepsilon))$ time.

The computation of z^A as the best possible combination of large and small items can be performed by going through y and S only once after sorting both sets. After scaling y contains only $O(1/\varepsilon^2)$ entries. The $O(n \log n)$ factor caused by sorting S can be replaced by $O(n \log(1/\varepsilon))$ following an iterative median strategy by Lawler [295, Section 6]. Thus, the running time of algorithm FPKP can be stated as

$$O(n \cdot \min\{\log n, \log(1/\varepsilon)\} + 1/\varepsilon^2 \log(1/\varepsilon) \cdot \min\{n, 1/\varepsilon \log(1/\varepsilon)\})$$

plus the effort caused by Recursion.

To analyze the running time of Recursion we refer again to the representation of the recursive structure as a binary tree as introduced in the proof of Lemma 6.2.7 above. Since the resulting calculation of the running time spent in the recursion follows closely the general description in Section 3.3 and was already carried out in detail for the more complicated *FPTAS* for (SSP) in Section 4.6 in the proof of Theorem 4.6.8, we refrain from repeating the required arguments. It is sufficient to mention that the running time of the two executions of Interval-Dynamic-Programming in level ℓ of the recursion are bounded with Corollary 6.2.9 by

$$O(D(L)/2^\ell \cdot \tilde{q}/(z^\ell \varepsilon^2) \cdot \log(\tilde{q}/(z^\ell \varepsilon))).$$

If a node i in level ℓ has an input profit $\tilde{q}_\ell^i$ we sum up over all m_ℓ nodes in a level and over all levels reaching a total computation time for all nodes of

$$\begin{aligned} \sum_{\ell=0}^{\log D(L)} \sum_{i=1}^{m_\ell} \frac{D(L)}{2^\ell} \frac{\tilde{q}_\ell^i}{z^\ell \varepsilon^2} \log\left(\frac{\tilde{q}_\ell^i}{z^\ell \varepsilon}\right) &\leq \sum_{\ell=0}^{\infty} \frac{D(L)}{2^\ell} \cdot 2/\varepsilon^2 \cdot \log(2/\varepsilon) \\ &\leq 2D(L) \cdot 2/\varepsilon^2 \cdot \log(2/\varepsilon). \end{aligned}$$

This is of order $O(1/\varepsilon^2 \log(1/\varepsilon) \cdot \min\{n, 1/\varepsilon \log(1/\varepsilon)\})$, which means that the recursive structure does not cause an increase in asymptotic running time.

After Scaling-Reduction the dynamic programming arrays y and r clearly require $O(1/\varepsilon^2)$ space.

Since the total length required for the arrays during the recursion always adds up to $O(1/\varepsilon^2)$ we can always work on the same memory positions by an efficient handling of "free" space. The corresponding step $(*)$ in Recursion should keep track of available parts of the initial memory position. An alternative to this diligent memory management would be to recompute y_2 and r_2 after returning from the recursion on $\tilde{L}_1$ by repeating Interval-Dynamic-Programming $(\tilde{L}_2, \tilde{q})$ and thus to be able to use these memory positions in Recursion $(\tilde{L}_1, q_1 - z^N)$.

The recursive bipartitions of $D(\tilde{L})$ in Recursion can be handled without using additional memory e.g. by partitioning $\tilde{L}$ into a set with smaller profits and one with larger profits. In this case every subset is a consecutive interval of indices and can be referenced by its endpoints.

Hence, every execution of Recursion requires only a constant amount of additional memory. As the recursion depth is bounded by $O(\log(1/\varepsilon))$, the overall space bound follows. □

Under the usual assumption that n is of much larger magnitude than $1/\varepsilon$ (cf. Lawler [295]) the running time of FPKP given in (6.7) can be stated simply as

$$O(n \log(1/\varepsilon) + 1/\varepsilon^3 \log^2(1/\varepsilon)). \tag{6.8}$$

Concerning the practical behaviour of FPKP it can be noted that its performance hardly depends on the input data and is thus not sensitive to "hard" (KP) instances. Furthermore, it can be expected that the tree-like structure implied by Recursion (see Lemma 6.2.7) will hardly be fully generated thanks to the partial solutions delivered by Backtracking. Exploiting the previously mentioned possible reduction of L after every execution of Backtracking the maximum recursion depth should be fairly moderate in practice.

7. The Bounded Knapsack Problem

Right from the beginning of research on the knapsack problem in the early sixties separate considerations were devoted to problems where a number of *identical copies* of every item are given or even an unlimited amount of each item is available. The corresponding problems are known as the *bounded* and *unbounded knapsack problem*, respectively. Since there exists a considerable amount of theoretical, algorithmic and computational results which apply for only one of these two problems, we found it appropriate to deal with them in separate chapters. Hence, we restrict this chapter to the bounded case whereas the unbounded case will be treated in Chapter 8.

The chapter will start with an introduction including the transformation to a standard knapsack problem. Several interesting contributions concerning dynamic programming algorithms are described in Section 7.2. Branch-and-bound methods together with computational results will be given in Section 7.3 followed by approximation algorithms which will be the subject of Section 7.4.

7.1 Introduction

Let us recall from Chapter 1 the precise definition of the problem under consideration. A set of *item types* $N := \{1,\dots,n\}$ is given where all items of type j have profit p_j and weight w_j. There are b_j identical copies of item type j available. The corresponding integer programming formulation of the bounded knapsack problem (BKP) is given as follows.

$$\text{(BKP)} \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{7.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c, \tag{7.2}$$

$$0 \le x_j \le b_j,\ x_j \text{ integer}, \quad j = 1,\dots,n. \tag{7.3}$$

Clearly, the selection of objects under a capacity constraint where a limited number of identical copies are available for every object type is a very natural model for

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

many practical decision problems. Classical applications are given in the computation of the most profitable loading of a ship (see Gilmore and Gomory [177]) or equivalently the most valuable loading of the space shuttle (see Render and Stair [405]), and for cutting problems in one, two or more dimensions (see Gilmore and Gomory [176], Dyckhoff and Finke [120] and Section 15.1). In a business context investment decisions were considered, e.g. where a selection must be made between different research and development problems (see Bradley, Hax and Magnanti [45, ch. 9]), and general capital budgeting problems (see e.g. Weingartner [478] and Section 15.4).

In analogy to (1.14) we assume that

$$b_j \leq \left\lfloor \frac{c}{w_j} \right\rfloor, \quad j = 1,\ldots,n, \tag{7.4}$$

since no more than $\lfloor c/w_j \rfloor$ copies of item type j will fit into the knapsack. If this is not true for some item type j, one can immediately reduce the number of copies for this item type by setting $b_j := \lfloor c/w_j \rfloor$. Assumption (1.15), which excludes trivial problem instances where all items can be packed, translates into

$$\sum_{j=1}^{n} w_j b_j > c. \tag{7.5}$$

In the remainder of this chapter we will also refer to the LP-relaxation of (BKP) where the integrality condition for x_j is replaced by $x_j \in \mathbb{R}$ for all $j = 1,\ldots,n$. The optimal solution vector of the LP-relaxation for (KP) was given in Theorem 2.2.1. A straightforward analogon of the argument given there establishes the corresponding result for (BKP). If the item types are sorted in decreasing order of their efficiencies $e_j := p_j/w_j$ (see Section 2.1) then in analogy to (2.4) the *split item type* s is defined by

$$\sum_{j=1}^{s-1} w_j b_j \leq c \quad \text{and} \quad \sum_{j=1}^{s} w_j b_j > c. \tag{7.6}$$

Corollary 7.1.1. *If the item types are sorted by decreasing efficiencies such that*

$$e_1 \geq e_2 \geq \ldots \geq e_n,$$

then the optimal solution vector $x^{LP} := (x_1^{LP},\ldots,x_n^{LP})$ *of the LP-relaxation of (BKP) is given by*

$$\begin{aligned} x_j^{LP} &:= b_j \quad \text{for } j = 1,\ldots,s-1, \\ x_s^{LP} &:= \frac{1}{w_s}\left(c - \sum_{j=1}^{s-1} w_j b_j\right), \\ x_j^{LP} &:= 0 \quad \text{for } j = s+1,\ldots,n. \end{aligned}$$

A straightforward adaptation of Section 3.1 can be used to find the split item type s for (BKP) in $O(n)$ time. Also algorithm Greedy of Section 2.1 can be easily adapted to (BKP). In fact, it suffices to replace the packing of a single item into the knapsack by the packing of as many items as possible from a given item type. This quantity is bounded by the number of available copies b_j and by the remaining knapsack capacity. The resulting algorithm B-Greedy for (BKP) is given in Figure 7.1. Remarks on the performance of B-Greedy are given in Section 7.4.

Algorithm B-Greedy:

$\overline{w} := 0$ $\quad$ *$\overline{w}$ is the total weight currently packed*
$z^G := 0$ $\quad$ *z^G is the profit of the current solution*
for $j := 1$ **to** n **do**
  if $\overline{w} + w_j \leq c$ **then**
    put item j into the knapsack
    $x_j := \min\{b_j, \lfloor (c - \overline{w})/w_j \rfloor\}$
    $\overline{w} := \overline{w} + w_j x_j$
    $z^G := z^G + p_j x_j$
  else $x_j := 0$

Fig. 7.1. Greedy algorithm for (BKP).

Example: Consider an instance of (BKP) with capacity $c = 9$ and $n = 5$ item types with the following data:

j	1	2	3	4	5
p_j	10	5	7	6	1
w_j	3	2	3	4	1
b_j	2	3	2	4	5

The items are already sorted in decreasing order of efficiencies. By assumption (7.4), we can reduce b_4 from 4 to 2. The split item type is given by $s = 2$ and the optimal solution of the LP-relaxation by $x_1^{LP} := 2$, $x_2^{LP} := 1.5$ and $x_3^{LP} := x_4^{LP} := x_5^{LP} := 0$ with $z^{LP} = 27.5$. Algorithm B-Greedy puts both copies of item type 1 and one copy of type 2 into the knapsack. Finally, one copy of item type 5 can be added to fill the capacity completely. This yields $x_1^G := 2$, $x_2^G := 1$, $x_3^G := x_4^G := 0$ and $x_5^G := 1$ with solution value $z^G = 26$. The optimal solution would consist of two copies of item type 1 and one copy of item type 3 yielding an optimal solution value $z^* = 27$. □

7.1.1 Transformation of (BKP) into (KP)

Comparing the bounded knapsack problem with the binary problem (KP) one can obviously tackle (BKP) by defining an appropriate instance of (KP) which consists

of one distinct item for every copy of every item type in (BKP). Since every item type j gives rise to b_j items, the resulting instance of (KP) has $\sum_{j=1}^{n} b_j$ items. If the values b_j are only of moderate quantity, this straightforward approach is reasonable and all the methods presented in Chapter 5 can be applied. However, if $\sum_{j=1}^{n} b_j$ is much larger than n, this procedure would not yield reasonable solution methods or at least cause extremely unfavourable performances. Considering the upper bound on the number of items of type j given in (7.4), it can be seen that $\sum_{j=1}^{n} b_j$, i.e. the number of items in the (KP) instance, may be of order $O(nc)$, clearly a very unpleasant perspective.

A more efficient transformation can be constructed by coding the number of copies of every item type j in a binary format. Every integer number can be uniquely represented as a partial sum of the sequence of powers of 2, e.g. $b_j = 23 = 2^4 + 2^2 + 2^1 + 2^0$. Each of the powers of 2 appears at most once in such a sum, i.e. its selection is a binary decision. Therefore, we can model the variable x_j in (BKP) by binary variables each of them equal to a power of 2 times a single copy of this item type.

More formally, we introduce $n_j := \lfloor \log b_j \rfloor$ "artificial items" for every item type j in the corresponding (KP) such that the profits and weights are multiples of w_j and p_j by a power of 2. Recall that log always refers to the base 2 logarithm. Finally, we add another item with a multiplier $b_j - \sum_{i=0}^{n_j-1} 2^i = b_j - (2^{n_j} - 1)$ such that the profit and weight sum of all constructed items adds up exactly to $b_j p_j$, respectively $b_j w_j$. Note that by definition of n_j there is $2^{n_j} - 1 < b_j$. This means that we generate the following items:

profit	p_j	$2p_j$	$4p_j$	$8p_j$	...	$2^{n_j-1}p_j$	$(b_j - 2^{n_j} + 1)p_j$
weight	w_j	$2w_j$	$4w_j$	$8w_j$	...	$2^{n_j-1}w_j$	$(b_j - 2^{n_j} + 1)w_j$

The optimal solution of the resulting instance of (KP) is equivalent to the optimal solution of the original (BKP) since every possible value of $x_j \in \{0, \ldots, b_j\}$ can be represented by a sum of the above binary variables. On one hand every integer between 1 and $2^{n_j} - 1$ can be written as a partial sum of the values $1, 2, 4, \ldots, 2^{n_j-1}$. On the other hand by adding the last item to these sums every integer between $b_j - 2^{n_j} + 1$ and $(b_j - 2^{n_j} + 1) + 2^{n_j} - 1 = b_j$ can be reached but b_j will never be exceeded. Note that the values between $b_j - 2^{n_j} + 1$ and 2^{n_j-1} have a double representation. However, this redundancy does not increase the number of binary variables since any binary representation of b_j requires at least $\lfloor \log b_j \rfloor + 1$ elements.

Example: An illustration of this transformation for an instance of (BKP) with 3 item types and arbitrary profits and weights is given in Figure 7.2. The right half of the figure contains the corresponding instance of (KP). Note that for item type 2, two identical copies of item (p_2, w_2) are required. □

	(BKP)		(KP)					
item type 1:	$b_1 = 7$	$n_1 = 2$	profit	p_1	$2p_1$	$4p_1$		
			weight	w_1	$2w_1$	$4w_1$		
item type 2:	$b_2 = 16$	$n_2 = 4$	profit	p_2	$2p_2$	$4p_2$	$8p_2$	p_2
			weight	w_2	$2w_2$	$4w_2$	$8w_2$	w_2
item type 3:	$b_3 = 18$	$n_3 = 4$	profit	p_3	$2p_3$	$4p_3$	$8p_3$	$3p_3$
			weight	w_3	$2w_3$	$4w_3$	$8w_3$	$3w_3$

Fig. 7.2. Transformation of an instance of (BKP) with three item types into an instance of (KP) with 13 items.

An instance of (KP) generated by this transformation has $\sum_{j=1}^{n}(n_j + 1) = n + \sum_{j=1}^{n} \lfloor \log b_j \rfloor$ binary variables which is of order $O(n \log c)$, a significant improvement upon the previous bound of $O(nc)$. Indeed $O(n \log c)$ is polynomial in the input size whereas $O(nc)$ was only pseudopolynomial, a difference which will be crucial for the construction of approximation schemes in Section 7.4. However, the dependence on c is still a major drawback for the performance of practical algorithms.

Another difficulty of this transformation is the generation of items in (KP) with very large coefficients. Note that the largest weight coefficients in the constructed instance are close to $b_j w_j/2$ which will usually slow down considerably dynamic programming based algorithms. Furthermore, all items in the (KP) instance originating from the same item type in (BKP) clearly have the same efficiency e_j. However, instances with some correlation between profits and weights (see Section 5.5) were observed to be hard to solve in practice (see also Pisinger [385]). Therefore, it is worthwhile to deal more specifically with (BKP) and to develop algorithms which exploit the special structure of the problem.

Note that it is also possible to transform an instance of (BKP) into a multiple-choice knapsack problem (MCKP) as introduced in Section 1.2 and further treated in Chapter 11. In this case, each item type j corresponds to a class of $b_j + 1$ items, each of them representing one of the $b_j + 1$ possible choices (including 0) for the number of copies of item type j to be included in the knapsack. The profits and weights of these items are trivially given by the corresponding multiples of a single item. This transformation requires $\sum_{j=1}^{n}(b_j + 1)$ items which seems to be less attractive than the construction given above. However, the particular structure of the resulting (MCKP) instances may be taken into account and specialized variants of (MCKP) algorithms could be used for the solution of (BKP).

The transformation of (BKP) into (MCKP) is especially meaningful for the following variant of (BKP). In a production environment, it may happen that the inclusion of any positive number of copies of an item type j requires certain *setup operations* consuming resources v_j from the total capacity c. In addition, every copy of item

type j consumes w_j units of capacity, as before. This means that every item type causes fixed and variable costs. The resulting *bounded knapsack problem with setup costs* can be formulated as

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} (v_j y_j + w_j x_j) \leq c, \\ & 0 \leq x_j \leq b_j y_j, \; x_j \text{ integer}, \quad j = 1, \ldots, n, \\ & y_j \in \{0,1\}, \quad j = 1, \ldots, n. \end{aligned}$$

Obviously, the binary variable y_j indicates whether item type j is used or not, whereas x_j denotes the number of copies selected for every item type j.

Knapsack problems with setup costs analogous to this model were considered by Chajakis and Guignard [76] and Süral, van Wassenhove and Potts [448]. In the former paper, it is assumed that including items of type j and thus setting $y_j = 1$ also contributes a profit q_j. In the latter paper it is observed that this variant of (BKP) can be easily modeled by an instance of (MCKP). In fact, the same transformation as outlined above for (BKP) can be applied with the obvious modification that the item weights in the (MCKP) instance must include the setup costs of the corresponding item type.

7.2 Dynamic Programming

The basic dynamic programming approach for (KP) was described in detail in Section 2.3. Throughout the algorithm there are two options to be evaluated for every entry of the dynamic programming function, namely including an item in the knapsack or not. The better alternative is chosen. This concept can be extended in a straightforward way to (BKP) by not only considering whether to pack an item or not but how many items of a given item type should be packed, if any.

The resulting dynamic programming function $z_j(d)$ gives the optimal solution value of an instance of (BKP) restricted to the item types $1, \ldots, j$, $0 \leq j \leq n$, and capacity d with $0 \leq d \leq c$. As an extension of recursion (2.8) from Section 2.3 we can formulate:

$$z_j(d) := \max_{\ell=0,\ldots,b_j} \{z_{j-1}(d - \ell w_j) + \ell p_j \mid d \geq \ell w_j\} \tag{7.7}$$

Parameter ℓ denotes the number of copies of item type j to be included into a knapsack with capacity d. The evaluation of every function value by (7.7) may require the computation of the maximum over $b_j + 1$ expressions which consumes $O(c)$ running time. Thus a direct analogon of DP-2 would yield a dynamic programming

algorithm with an overall running time of $O(nc^2)$. Obviously, this algorithm also solves the all-capacities version of the problem.

Following the considerations of Section 2.3 the space requirements are $O(nc)$. In Section 3.3 a general procedure to reduce the storage of dynamic programming based algorithms was introduced. The main property for the use of this method is the possibility to compute only the optimal solution value with smaller storage requirements than the computation of the complete optimal solution vector. This property holds for the above dynamic programming scheme since the omission of storing the solution set for every value of $z_j(d)$ reduces the space requirements to $O(c)$. Hence, we can conclude with Theorem 3.3.1 that (BKP) can be solved in $O(n+c)$ space.

Note that the above time bound of $O(nc^2)$ should be compared to the transformation of (BKP) into a knapsack problem as described in Section 7.1.1. The solution of the corresponding instance of (KP) with $O(n \log c)$ items requires $O(nc \log c)$ time and $O(n \log c + c)$ space by Corollary 3.3.2. This is clearly superior to the above naive dynamic programming algorithm with respect to running time but not with respect to memory requirements. In Section 7.2.2 a more sophisticated dynamic programming algorithm will be given which dominates the transformation approach both in time and space. It solves (BKP) in $O(nc)$ time and $O(n+c)$ space which are the same complexities as currently known for (KP), except for the word RAM algorithm described in Section 5.2.1.

For later use in the discussion of an *FPTAS* for (BKP) in Section 7.4 we will introduce dynamic programming by profits. The notation and basic intuition follows the presentation of DP-Profits as given in Section 2.3. In particular, $y_j(q)$ denotes the minimal weight of a collection of copies from item types $1,\dots,j$ with total profit q. After computing an upper bound $U \geq z^*$ on the optimal solution value, e.g. by B-Greedy, and after the trivial initialization with $y_0(0) := 0$, $y_0(q) := c+1$ for $q = 1,\dots,U$, the dynamic programming function can be evaluated for $j = 1,\dots,n$ and $q = 0,\dots,U$ by the recursion

$$y_j(q) := \min_{\ell=0,\dots,b_j} \{y_{j-1}(q-\ell p_j) + \ell w_j \mid q \geq \ell p_j\}. \tag{7.8}$$

Without going into any improvement techniques the running time of this approach can be bounded by $O(\sum_{j=1}^{n} b_j U)$ which is $O(ncU)$. The space requirements are $O(nU)$ and can be improved with Section 3.3 to $O(n+U)$.

7.2.1 A Minimal Algorithm for (BKP)

A different application of dynamic programming with both theoretically measurable worst–case behaviour and very good practical performance was presented by Pisinger [388]. The algorithm is based on the core concept introduced in Section 5.4. In particular, the method pursues an expanding core strategy similar to

algorithm Minknap from Section 5.4.2. Reduction steps and other technical improvements are also included.

The main engine of the Bouknap algorithm is a dynamic programming algorithm which makes use of the fact that generally only a few variables x_j around s need to be changed from their LP-optimal values in order to obtain the optimal integer values. The recursion may be seen as a generalization of the recursion presented in Section 5.3 for the standard knapsack problem. Hence, assume that the items are ordered according to decreasing efficiencies, and let $z_{a',b'}(d), a' \leq s,\ b' \geq s-1,\ 0 \leq d \leq 2c$, be an optimal solution to the core problem:

$$z_{a',b'}(d) := \sum_{j=1}^{a'-1} b_j p_j + \left\{ \max \sum_{j=a'}^{b'} p_j x_j \;\middle|\; \begin{array}{l} \sum_{j=a'}^{b'} w_j x_j \leq d - \sum_{j=1}^{a'-1} b_j w_j, \\ x_j \in \{0,\ldots,b_j\} \text{ for } j = a',\ldots,b' \end{array} \right\}. \tag{7.9}$$

Notice that $z_{a',b'}(d)$ is an optimal solution to the bounded knapsack problem defined for items $a',\ldots,b'$ assuming that $x_j = b_j$ for $j < a'$ and $x_j = 0$ for $j > b'$. This leads to the following recursions:

$$z_{a',b'}(d) := \max \begin{cases} z_{a',b'-1}(d), \\ z_{a',b'-1}(d - w_{b'}) + p_{b'} & \text{if } d - w_{b'} \geq 0, \\ \quad\vdots \\ z_{a',b'-1}(d - b_{b'} w_{b'}) + b_{b'} p_{b'} & \text{if } d - b_{b'} w_{b'} \geq 0, \end{cases} \tag{7.10}$$

$$z_{a',b'}(d) := \max \begin{cases} z_{a'+1,b'}(d), \\ z_{a'+1,b'}(d + w_{a'}) - p_{a'} & \text{if } d + w_{a'} \leq 2c, \\ \quad\vdots \\ z_{a'+1,b'}(d + b_{a'} w_{a'}) - b_{a'} p_{a'} & \text{if } d + b_{a'} w_{a'} \leq 2c\,. \end{cases} \tag{7.11}$$

If $\overline{p}$ and $\overline{w}$ are the profit and weight sums of the split solution we may initially set $z_{s,s-1}(d) = \overline{p}$ for $d = \overline{w},\ldots,2c$ and $z_{s,s-1}(d) = -\infty$ for $d = 0,\ldots,\overline{w}-1$. Thus the enumeration starts at $(a',b') = (s,s-1)$ and continues by alternating using recursion (7.10) to insert an item type b' into the knapsack, or using recursion (7.11) to remove an item type a' from the knapsack. An optimal solution to (BKP) is found as $z_{1,n}(c)$.

The Bouknap algorithm may now be described as illustrated in Figure 7.3. The algorithm is using DP-with-Lists as described in Section 3.4, fathoming states which provably cannot lead to an optimal solution. The dynamic programming algorithm in step 2 is combined with upper bound tests as described in Section 3.5, using an upper bound U to fathom unpromising states.

Since the time complexities of recursion (7.10) and (7.11) are proportional to the bound b_j on an item type j, a bounding test is used for tightening the bound. In this way the bound may be restricted to $x_j \in \{0,\ldots,m_j\}$ if $j \geq s$, or to $x_j \in \{b_j - m_j,\ldots,b_j\}$ if $j < s$. Thus assume that the core is $C = \{a,\ldots,b\}$ and that we are going to add item type j to the core, where $j = a' - 1$ or $j = b' + 1$.

Algorithm Bouknap:

1. Find an initial core $C = \{s\}$ through a version of Find-Core pushing all discarded intervals into two stacks H and L.
2. Run the dynamic programming recursion alternately inserting an item type b' by (7.10) or removing an item type a' by (7.11). For each step, check if a' and b' are within the current core C, otherwise pick the next interval from H or L, reduce and sort the items, add the remaining items to the core.
3. Before an item type is processed by dynamic programming: Test (7.13) and (7.14) to tighten the bound on the item type.
4. Stop, when all states in the dynamic programming have been fathomed due to an upper bound test, or all items $j \notin C$ have been fixed at their optimal value.

Fig. 7.3. Algorithm Bouknap following an expanding core strategy to run dynamic programming only for the core problem.

It is only fruitful to insert m items of type $j = b' + 1$ in the knapsack if an upper bound U on (BKP) with additional constraint $x_j = m$ exceeds the current lower bound z^ℓ. For this purpose we use a generalization of the Dembo and Hammer bound [104] defined in (5.29), obtaining the test

$$\overline{p} + mp_j + (c - \overline{w} - mw_j)\frac{p_s}{w_s} \geq z + 1. \tag{7.12}$$

From this inequality we may obtain the *maximum number* m_j^+ of copies of item type j which may be included in the knapsack:

$$m_j^+ = \left\lfloor \frac{w_s(z^\ell + 1 - \overline{p}) - p_s(c - \overline{w})}{p_j w_s - p_s w_j} \right\rfloor \text{ for } j = b' + 1, \tag{7.13}$$

where we set $m_j^+ = b_j$ if $p_j/w_j = p_s/w_s$ or when the right side of equation (7.13) is larger than b_j. A similar result is obtained for items of type $j = a' - 1$, which have to be removed from the knapsack. Here an upper bound m_j^- on the number of removed copies of item type j is given by

$$m_j^- = \left\lfloor \frac{w_s(z^\ell + 1 - \overline{p}) - p_s(c - \overline{w}))}{p_s w_j - p_j w_s} \right\rfloor \text{ for } j = a' - 1, \tag{7.14}$$

with the same conventions as for equation (7.13).

Although these bounds work well for most instances, tighter reductions may be derived by using enumerative bounds as presented in Pisinger [383] and generalized to the (BKP) in [388].

In a similar way as in Theorem 5.4.2 it can be shown that the enumerated core of Bouknap is minimal. The worst-case running time of Bouknap is still bounded by $O(nc^2)$ although the practical behaviour can be expected to be far below this upper bound.

7.2.2 Improved Dynamic Programming: Reaching (KP) Complexity for (BKP)

The straightforward dynamic programming recursion (7.7) runs in $O(c)$ time for each of the $c+1$ function entries and for every item type. This is the same computational effort per item type as required for computing in constant time the corresponding values of the dynamic programming function in the knapsack problem attained by the trivial transformation of every item type into $O(c)$ identical items. From this point of view the special structure of (BKP) did not help at all to gain solution methods with better performance. In this section we will describe a more advanced dynamic programming algorithm due to Pferschy [374] which heavily makes use of the presence of multiple items of the same type. Note that the presented algorithm also solves the all-capacities version of (BKP).

Let us first give an informal description of the main idea of the algorithm. In principle, the approach is connected to the transformation into a knapsack problem since every copy of an item type is added separately to an existing partial solution. However, the copies of the same item type are treated all together in an efficient way and not in independent iterations.

In particular, as in the classical dynamic programming approach, after processing all item types $1,\dots,j-1$ we try to add a first item of type j to every solution set attached to an entry of the dynamic programming function. If the corresponding comparison of the function value $z_{j-1}(d+w_j)$ with $z_{j-1}(d)+p_j$ induces an update, i.e. $z_j(d+w_j)=z_{j-1}(d)+p_j$, we immediately add a second copy to $z_{j-1}(d)$ and thereby try to improve $z_{j-1}(d+2w_j)$. In this way we proceed as long as there are items available (at most b_j) and as long as updates, i.e. improvements of a function value, take place. By this procedure a *sequence of updates* starting at d is generated.

If this sequence is terminated because all b_j copies were successfully added, a new update sequence is started to (possibly) improve function values for a capacity value higher than $d+b_jw_j$. Clearly, no further update can be attained for values $d+\ell w_j$ for $\ell=2,\dots,b_j$ by adding items of type j to any entry $z_{j-1}(d')$ for $d'>d$.

The most promising starting position for an update of values larger than $d+b_jw_j$ lies somewhere between $d+w_j$ and $d+b_jw_j$ and can be computed efficiently by keeping track of the necessary information during the previous update sequence. If the update sequence was stopped because at some point $d+\ell w_j$ with $1\le\ell\le b_j$ no update took place, i.e. $z_j(d+\ell w_j)=z_{j-1}(d+\ell w_j)$, then no further updates from starting points between $d+w_j$ and $d+(\ell-1)w_j$ are worth to be considered either

since it will always be better to add a copy of item type j to the "dominating" function entry at $d + \ell w_j$.

To follow this idea of update sequences in the algorithm it is useful to consider the function entries for every new item type j not consecutively, i.e. by incrementing the capacity value always by 1, but to combine those with the same remainder of a division by w_j. This means that the capacities are considered in groups, the first one consisting of $0, w_j, 2w_j, \ldots$, the second one of $1, 1+w_j, 1+2w_j, \ldots$ and so on. A related strategy was used for items with identical weights but different profits in algorithm FPKP for (KP) in Section 6.2.3.

It remains to be discussed how to determine efficiently the best starting position $d+\ell' w_j$ with $1 \leq \ell' \leq b_j$ for a new update sequence after performing b_j consecutive updates from capacity d. It can be easily checked that the most promising starting position to possibly induce an update of $z_j(d+(b_j+1)w_j)$ by adding the appropriate number of copies of item type j is determined by

$$\ell' := \arg\max\{z_{j-1}(d+\ell w_j) + (b_j+1-\ell)p_j \mid 1 \leq \ell \leq b_j\}. \tag{7.15}$$

Since $(b_j+1)p_j$ and dp_j/w_j are independent from ℓ, this can be written equivalently as

$$\ell' = \arg\max\{z_{j-1}(d+\ell w_j) - p_j/w_j(d+\ell w_j) \mid 1 \leq \ell \leq b_j\}. \tag{7.16}$$

Considering the dynamic programming function geometrically in a (*weight, profit*) space this can be seen as minimizing the distance in profit direction between the point $(d+\ell w_j, z_{j-1}(d+\ell w_j))$ and the line through the origin with ascent $e_j = p_j/w_j$. However, these distances can be computed well before the update operations for item type j are started at all. In particular, it will be convenient to store the values in an auxiliary array

$$\delta(d) := d\,p_j/w_j - z_{j-1}(d) \quad \text{for } d = 0, \ldots, c. \tag{7.17}$$

The situation is further illustrated in Figure 7.4.

To compute this minimum distance we would like to avoid going through all b_j candidates. In particular, we should not check the same values several times, which may be necessary in a naive approach since different starting values d close to each other give rise to overlapping search intervals $d+w_j$ to $d+b_j w_j$. Therefore we already keep track of the capacity values which are most promising for starting further update sequences during the generation of the updates. Beside storing the capacity position d' with minimum distance it is useful to also store the position with minimum distance among all capacity values larger than d' because this may give rise to a future update sequence. Continuing this strategy we take all capacity values encountered during an update sequence and insert them into a sorted list. This list is organized to contain an increasing sequence of capacity values with increasing distances δ. In case of a tie, i.e. identical values of δ, a higher capacity is preferred since it offers chances for a longer sequence of updates.

Hence, the following operations on a list L are required:

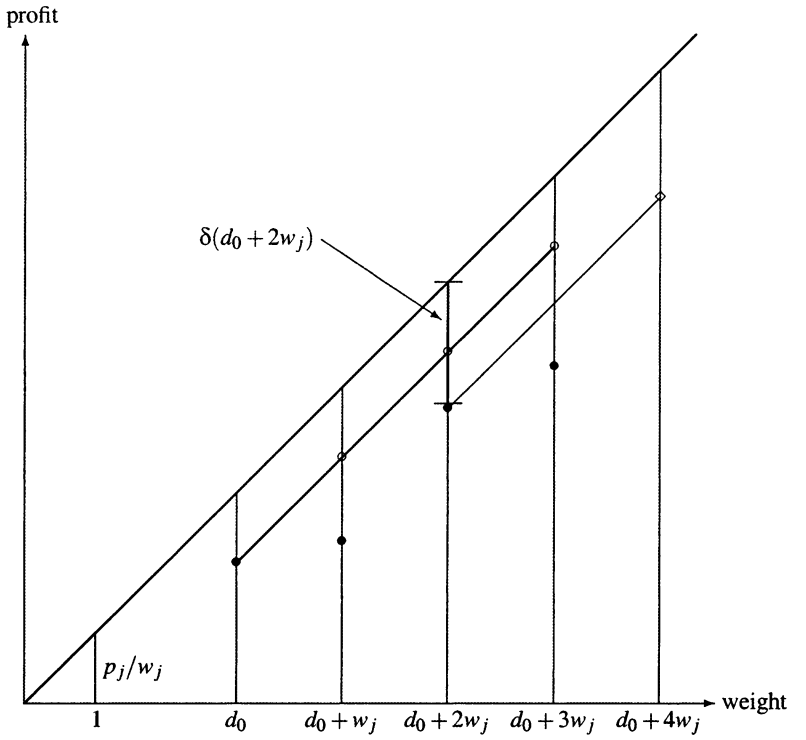


Fig. 7.4. Construction of the starting point for the next possible update ($b_j = 3$): "•" are values of $z_{j-1}(\cdot)$, "○" are updated values of $z_j(\cdot)$. The point corresponding to $z_{j-1}(d_0+2w_j)$ has the smallest distance $\delta(d_0+2w_j)$ to the line through the origin with ascent $e_j = p_j/w_j$ and hence is the best starting point for the next possible update yielding "◇" as a candidate for $z_j(d_0+4w_j)$.

insert capacity d with value $\delta(d)$ in list L :
Start from the end of the list and move forward towards the beginning. If the previous entry has a value of $\delta(\cdot)$ larger than or equal to $\delta(d)$, delete it and move on. If it has a smaller value, add d as last element in the list. In this way, the entries of L are maintained in increasing order both of d and of $\delta(d)$.

getmin L :
Return the first element in list L, i.e. the capacity d with the smallest $\delta(d)$, and delete it from the list.

delete L : Delete all entries of list L.

An explicit description of the resulting algorithm Improved-DP is given in Figure 7.5. To keep the presentation simple the manipulation of the subsets of items corresponding to every dynamic programming function entry is omitted. However, all considerations about storing solution vectors as performed in Section 2.3 apply in an analogous way.

```
Algorithm Improved-DP:
z_j(d) := 0 for d = 0,...,c and j = 0,...,n        initialization
sort the item types 1,...,n in increasing order of efficiencies p_j/w_j
for j := 1 to n do
    for d := 0,...,c do
        δ(d) := d p_j/w_j − z_{j−1}(d)
        Note that δ(d) ≥ 0 for all d after sorting set N
    for r := 0,...,w_j − 1 do        all residual values of c/w_j
        d_0 := r
        d_0 is the starting capacity for a sequence of possible
        consecutive updates
        d := r + w_j      d is the capacity for a considered update operation
        ℓ := 1      ℓ copies of item type j are added to capacity d_0
        delete L
        while d ≤ c do
            if z_{j−1}(d) < z_{j−1}(d_0) + ℓ p_j then        update
                z_j(d) := z_{j−1}(d_0) + ℓ p_j
                add a copy of item type j
                insert d in L
                d := d + w_j
                if ℓ < b_j then ℓ := ℓ + 1
                further update in this sequence may be possible
                else      (ℓ = b_j)
                    d_0 := getmin L
                    ℓ := (d − d_0)/w_j
            else
                z_j(d) := z_{j−1}(d)        no update
                d_0 := d, ℓ := 1
                d := d + w_j
                delete L
            end if
        end while
    end for r
end for j
```

Fig. 7.5. Algorithm Improved-DP performs dynamic programming for (BKP) by combining updates into sequences of consecutive operations.

To illustrate the course of algorithm Improved-DP we will give a numerical example taken from [335].

j	r	d_0	d	ℓ	$z_j(d)$	L	comment
1	0	0	5	1	11	5	$p_1 = 11, w_1 = 5$
			10	2	22	5,10	
..			...				

d	0	1	2	3	4	5	6	7	8	9	10
$z_1(d)$	0	0	0	0	0	11	11	11	11	11	22

j	r	d_0	d	ℓ	$z_j(d)$	L	comment
2	0	0	3	1	15	3	$p_2 = 15, w_2 = 3$
			6	2	30	3,6	
			9	3	45	3,6,9	
	1	1	4	1	15	4	
			7	2	30	4,7	
			10	3	45	4,7,10	
	2	2	5	1	15	5	
			8	2	30	5,8	

d	0	1	2	3	4	5	6	7	8	9	10
$z_2(d)$	0	0	0	15	15	15	30	30	30	45	45

j	r	d_0	d	ℓ	$z_j(d)$	L	comment
3	0	0	1	1	10	1	$p_3 = 10, w_3 = 1$
			2	2	20	1,2	
			3	3	30	1,3	$\delta[3] = 15 < 20 = \delta[2]$
			4	4	40	1,3,4	
			5	5	50	1,3,4,5	$b_3 = 6$
			6	6	60	1,3,4,6	$\delta[6] = 30 < 35 = \delta[5]$
		1	7	6	60	3,4,6,7	
		3	8	5	65	4,6,7,8	
			9	6	75	4,6,7,9	$\delta[9] = 45 < 50 = \delta[8]$
		4	10	6	75	6,7,9,10	

d	0	1	2	3	4	5	6	7	8	9	10
$\delta[d]$	0	10	20	15	25	35	30	40	50	45	55
$z_3(d)$	0	10	20	30	40	50	60	60	65	75	75

Table 7.1. Numerical example for the execution of Improved-DP: Every line gives the values of the variables just before reaching the command $d := d + w_j$ in the middle of the while loop.

Example: An instance of (BKP) with $n = 3$ and $c = 10$ is given by

$$\begin{array}{lll} p_1 = 11, & p_2 = 15, & p_3 = 10, \\ w_1 = 5, & w_2 = 3, & w_3 = 1, \\ b_1 = 2, & b_2 = 4, & b_3 = 6. \end{array}$$

In Table 7.1 we list the values of the variables throughout the execution of Improved-DP yielding the optimal solution $x_1^* = 0, x_2^* = 1, x_3^* = 6$ with value $z_3(10) = 75$. □

Theorem 7.2.1 Improved-DP *solves (BKP) in $O(nc)$ time and $O(nc)$ space.*

Proof. We will sketch only briefly why the algorithm solves (BKP) to optimality and then deal with the running time and space analysis.

Correctness:

In the classical dynamic programming approach for (BKP) defined in (7.7) $z_j(\cdot)$ is computed from $z_{j-1}(\cdot)$ by considering the possibility to add zero, one or up to b_j copies of item type j to every solution represented in $z_{j-1}(\cdot)$ and updating the corresponding entry in $z_j(\cdot)$ if a higher profit can be attained for some capacity.

Naturally, the order of checking all possible updates for one item type does not influence the correctness of this approach. Therefore, the strategy of considering all capacities with the same remainder r with respect to division by w_j in one loop does not change the overall construction.

Furthermore, it can be seen easily that if an update occurs by adding profit p_j to some entry $z_{j-1}(d')$, it will always be more promising to add another copy of item type j to the new entry $z_j(d' + w_j)$ than to the dominated old entry $z_{j-1}(d' + w_j)$. Therefore, optimality is not lost by trying to generate a sequence of consecutive updates until the upper bound b_j is reached. If this is the case, the most promising entry between $z_{j-1}(d_0 + w_j)$ and $z_{j-1}(d_0 + b_j w_j)$ to continue the search for a possible update is found as described above in detail.

Running Time:

Clearly, the running time is dominated by the executions of the loop over all remainders r for every item type j. Each of the w_j executions of the loop takes at most $O(c/w_j)$ time, because whenever we go to the beginning of the **while** loop we have increased the value d by w_j and we exit this loop as soon as $d + w_j$ is larger than the capacity c.

The main **if–then–else** statement in the **while** loop can be performed in constant time with the possible exception of the **insert** command. However, every position $d_0 + \ell w_j$ can be inserted in L and deleted again during another **insert** operation at most once. Hence, also the overall time for all **insert** operations during one loop is $O(c/w_j)$ time. Summing up over all remainders r and all item types yields the overall running time of $O(nc)$.

Note that the sorting of the items is done only for convenience to guarantee that all differences $\delta(d)$ are nonnegative, but it is not required for the correctness of the algorithm.

Space Requirements:

As mentioned above the storing of the solution vectors during dynamic programming was neglected in the algorithm for reasons of clarity. However, it is straightforward to store the corresponding subsets of items for every $z_j(d)$. As each subset representation can be done in $O(n)$ space, the overall space bound of $O(nc)$ follows.

□

Theorem 7.2.2 *(BKP) can be solved in $O(nc)$ time and $O(n + c)$ space.*

Proof. It is sufficient to show that Improved-DP fulfills conditions (1) to (4) of Section 3.3. Then the claim follows immediately from Theorem 3.3.1.

Performing Improved-DP without storing the solution vectors at all yields a procedure fulfilling the requirements of condition (1). Conditions (2) and (3) trivially hold and also (4) can be guaranteed by a simple set union operation. □

7.2.3 Word RAM Algorithm

Exploiting word-parallelism as described in Section 3.7, Pisinger [392] showed that we may obtain a somehow similar time complexity and a somehow improved space bound compared to Theorem 7.2.2.

Corollary 7.2.3 *The (BKP) can be solved in time $O(nm)$ and space $O(n+m/\log m)$ on a word RAM, where $m := \max\{c, z^*\}$.*

Proof. Transforming the (BKP) into a (KP) as described in Section 7.1.1 we obtain an instance of (KP) with $n' = \sum_{j=1}^{n} \lfloor \log b_j \rfloor + n$ items, which is of magnitude $O(n \log c)$. Using the Word RAM algorithm described in Section 5.2.1 to solve the resulting (KP) we obtain a running time of $O(n'm/\log m)$ which is in $O(nm)$. Since we do not need to store explicitly the transformed instance the space bound is $O(n+m/\log m)$. □

For the special case of a *bounded subset sum problem*, i.e. the case where $p_j = w_j$ for $j = 1,\ldots,n$ we may use the algorithm from Section 4.1.1 to solve the resulting (SSP), hence getting an overall running time of $O(nc)$ and a space bound of $O(n + c/\log c)$.

7.2.4 Balancing

Using the concept of balancing described in Section 3.6 we may reach a dynamic programming algorithm which runs in linear time for instances where the profits p_j, weights w_j and number of copies b_j are bounded by a constant. Defining $b_{\max} := \max\{b_j \mid j = 1,\ldots,n\}$, Pisinger [387] showed the following:

Corollary 7.2.4 *The (BKP) can be solved in $O(np_{\max}w_{\max}b_{\max})$ time and space.*

Comparing the above result to the time complexity $O(nc)$ from Theorem 7.2.2 we notice that c may be of magnitude $nw_{\max}b_{\max}$. In this case $O(nc)$ becomes of magnitude $O(n^2w_{\max}b_{\max})$. This illustrates, that balancing becomes attractive only for large values of c and n.

7.3 Branch-and-Bound

7.3.1 Upper Bounds

Upper bounds for branch-and-bound algorithms are usually derived in an analogous way as for (KP) in Section 5.1.1. Let us assume that the item types are sorted by decreasing efficiencies. Then the rounded down solution of the LP-relaxation yields the following trivial bound.

$$U_1 := U_{\mathrm{LP}} = \lfloor z^{LP} \rfloor = \left\lfloor \sum_{j=1}^{s-1} p_j b_j + \left(c - \sum_{j=1}^{s-1} w_j b_j \right) \frac{p_s}{w_s} \right\rfloor . \tag{7.18}$$

An improved upper bound can be attained by a generalization of (5.12). Therefore, we consider two possibilities of filling the residual capacity $c_r := c - \sum_{j=1}^{s-1} w_j b_j$ depending on how many copies of the split item type s are packed. At most $\beta_s := \left\lfloor \frac{c_r}{w_s} \right\rfloor$ such copies can be added to the greedy solution containing all items of types 1 to $s-1$. Either there are exactly β_s copies packed and the remaining capacity is filled by item type $s+1$ or there are $\beta_s + 1$ items of type s put into the knapsack. In this case the consumed additional capacity reduces the contribution of item type $s-1$. Together this yields the following bound by Martello and Toth [323].

$$U_2 := \max \left\{ \left\lfloor \sum_{j=1}^{s-1} p_j b_j + p_s \beta_s + (c_r - w_s \beta_s) \frac{p_{s+1}}{w_{s+1}} \right\rfloor , \right. \tag{7.19}$$
$$\left. \left\lfloor \sum_{j=1}^{s-1} p_j b_j + p_s (\beta_s + 1) + (c_r - w_s (\beta_s + 1)) \frac{p_{s-1}}{w_{s-1}} \right\rfloor \right\}$$

Also more advanced bounds based on partial enumeration analogous to (5.13) can be obtained (see Pisinger [388]). Moreover, all bounds for (KP) can be applied to the instance attained by the transformation of (BKP) into the corresponding binary knapsack problem (KP) given in Section 7.1.1.

An interesting detail can be observed in the application of *maximum cardinality constraints*. Along the same lines as in (5.14) and (5.15) we can introduce an upper bound on the number of items in the optimal solution. Sorting the item types by *increasing weights* $w_1 \leq w_2 \leq \ldots \leq w_n$ and setting $t := \min\{h \mid \sum_{j=1}^{h} w_j b_j > c\}$ we define

$$k := \sum_{j=1}^{t-1} b_j + \left\lfloor \frac{c - \sum_{j=1}^{t-1} w_j b_j}{w_t} \right\rfloor . \tag{7.20}$$

Now we can add the following constraint to (BKP) without reducing the set of feasible solutions but (hopefully) reducing the optimal solution value of the LP-relaxation.

$$\sum_{j=1}^{n} x_j \leq k \tag{7.21}$$

It was observed in Pisinger and Toth [394] that (7.21) may yield a lower value of the LP-relaxation than the addition of the cardinality constraint (5.15) to the (KP) instance resulting from the transformation of (BKP). This can be shown by the following example.

Example: Consider an instance of (BKP) with two item types given by
$p_1 = 10, \quad p_2 = 11,$
$w_1 = 10, \quad w_2 = 11,$
$b_1 = 3, \quad b_2 = 3.$

The capacity c is set to 40. Evaluating (7.20) yields $k = 3$. It can be seen immediately that the optimal solution of the LP-relaxation of (BKP) with the additional constraint $\sum_{j=1}^{n} x_j \leq 3$ consists of 3 copies of item type 2 and has an optimal solution value of 33. Transforming the given instance into a (KP) according to Section 7.1.1 requires four items with the following data:

profit	10	20	11	22
weight	10	20	11	22

The cardinality constraint (5.14) yields $k = 2$. Adding the corresponding constraint to the LP-relaxation gives an optimal solution value of 40 following from the capacity constraint. □

7.3.2 Branch-and Bound Algorithms

During the seventies many papers appeared on specialized branch-and-bound methods for various integer programming problems related to the knapsack family. A few of them dealt explicitly with (BKP). However, we are not aware of any relevant publication on this topic since 1979. It seems that further research was more or less stopped by the observation that these branch-and-bound algorithms customized for (BKP) were generally outperformed by the application of branch-and-bound methods to the instance of (KP) derived by the transformation of Section 7.1.1. Computational experiments in this direction were reported by Martello and Toth in [335] where a branch-and-bound code for the (KP) instance was compared with the algorithms from [246] and [323] described in the following. Therefore, we will only mention briefly some of the historic papers with major significance.

Ingargiola and Korsch [246] (with corrections in [325]) extended their reduction method for (KP) discussed in Section 5.1.3 to the (BKP) case. This approach tries to fix a number of variables to 0 or 1 before running a branch-and-bound algorithm. Their reduction algorithm tries to diminish the range of the variables from $0, \ldots, b_j$

to a smaller interval. Thus, any branch-and-bound method will perform better on the reduced instance. In [246] a so-called branch-search algorithm related to a corresponding approach for (KP) as given by Greenberg and Hegerich [197] is applied.

Bulfin, Parker and Shetty [57] constructed and tested a branch-and-bound method composed of tools from more general papers of this period. In particular, they improved the upper bound in every node of the search tree by adding a penalty function which uses information from the optimal simplex tableau of the LP-relaxation of (BKP).

A modification of Algorithm MT1 (referred to in Section 5.1.4) for (BKP) was presented by Martello and Toth in [323] (see also the correction in [7]). A simplified description of this algorithm MTB is given in Figure 7.6.

Algorithm MTB($j,\overline{p},\overline{w}$):

```
      improved := false
1     if (w̄ > c) then return improved
2     if (p̄ > z^ℓ) then
3         z^ℓ := p̄, improved := true
4     else improved := false
5     if (j > n) or (c − w̄ < w_j) or (U_2 < z^ℓ + 1) then return improved
6     for a := b_j downto 0 do
7         if MTB(j+1, p̄ + ap_j, p̄ + aw_j) then
8             x_j := a, improved := true
9     return improved
```

Fig. 7.6. Recursive branch-and-bound algorithm MTB for (BKP).

It is assumed that the items are sorted according to decreasing efficiencies. The table $\underline{w}$ is given by $\underline{w}_j := \min_{i \geq j} w_i$. All decision variables x_j must be initialized to 0, and the lower bound set to $z^\ell = 0$ before calling MTB(0,0,0).

The algorithm is a recursive algorithm, which at each level $j = 0,\ldots,n$ assigns a value a to the variable x_j. Following a greedy-type approach, the algorithm considers values of a in the order $m, m-1,\ldots,0$. The parameters of MTB are as follows: j is the index of the variable to branch at, while $\overline{p} = \sum_{i=1}^{j} p_i x_i$ and $\overline{w} = \sum_{i=1}^{j} w_i x_i$.

Line 1 of the algorithm tests feasibility of the current solution. If $\overline{w} > c$, we may immediately backtrack. Knowing that the solution is feasible in line 2, we check whether the lower bound has been improved. Finally, line 5 checks whether any more items fit into the knapsack, and also evaluates the upper bound tests. The bound U_2 is derived by equation (7.19) restricted to variables $i \geq j$ and with reduced capacity $d = c - \overline{w}$.

The boolean variable improved is used to indicate whether an improved solution has been found at the present level, or any descendant nodes. If the variable is true (and only in this case), the solution vector x will be updated upon backtracking.

Martello and Toth [335] in a later paper presented a *fixed-core* branch-and-bound algorithm called MTB2. The algorithm is based on the transformation of (BKP) into an equivalent (KP) as described in Section 7.1.1. The transformed problem is then solved by the MT2 algorithm given in Section 5.4.2. One can argue whether MTB2 is a core-algorithm, since it transforms *all* items to an equivalent (KP) instance. It is however not difficult to modify the concept of MT2 to handle (BKP) instances directly, by making use of the MTB branch-and-bound algorithm. Instead of calling MT2 to solve the transformed problem, one can obviously use any well-performing algorithm for the (KP).

7.3.3 Computational Experiments

In order to compare specialized approaches for (BKP) with those based on a transformation to (KP), we have chosen to focus on the two most promising algorithms Bouknap (see Section 7.2.1) and MTB2 (see Section 7.3.2). The latter was tested in two versions: The published code MTB2 which solves the resulting instance of (KP) through algorithm MT2, and a new technique which solves (KP) through the MThard code. Following Martello and Toth's naming tradition we will call the last approach MTBhard.

The following data instances are considered:

- *Uncorrelated data instances*: p_j and w_j are randomly distributed in $[1,R]$.
- *Weakly correlated data instances*: w_j randomly distributed in $[1,R]$ and p_j randomly distributed in $[w_j - R/10, w_j + R/10]$ such that $p_j \geq 1$.
- *Strongly correlated data instances*: w_j randomly distributed in $[1,R]$ and $p_j = w_j + 10$.
- *Subset sum data instances*: w_j randomly distributed in $[1,R]$ and $p_j = w_j$.

The *data range* R will be tested with values $R = 100, 1000$ and 10000, while the bounds m_j are randomly distributed in $[5, 10]$.

Each problem type is tested with a series of 200 instances, such that the capacity c of the knapsack varies from 1% to 99% of the total weight sum of the items. This approach takes into account that the computational times may depend on the chosen capacity. In the following tables a dash means that the 200 instances could not be solved in a total time of 5 hours. The results have been obtained on a INTEL PENTIUM 4, 1.5 GHZ processor.

The computational times are presented in Table 7.2. The MTB2 algorithm has substantial stability problems for low-ranged data instances even of relatively small size. Subset sum data instances also show a few anomalous occurrences. The MTBhard algorithm generally has a more stable behavior, but it is interesting to note, that although MThard is excellent at solving strongly correlated (KP) instances, it is not able to solve strongly correlated instances when these are transformed from

Algorithm	$n \setminus R$	Uncorrelated 100	1000	10000	Weakly corr. 100	1000	10000	Strongly corr. 100	1000	10000	Subset sum 100	1000	10000
	100	0	0	0	0	0	0	1	8	61	0	1	14
	300	0	0	0	0	0	1	3	28	281	0	1	13
Bouknap	1000	0	0	0	0	1	2	12	100	979	0	1	14
	3000	0	1	1	0	1	6	41	408	3959	0	2	14
	10000	1	1	4	1	2	9	154	1711	11972	1	2	14
	100	0	0	0	0	1	2	—	—	—	0	2	15
	300	1	1	1	0	3	6	—	—	—	0	—	3768
MTB	1000	1	2	4	2	5	24	—	—	—	0	1	9
	3000	2	5	13	5803	4	55	—	—	—	1	2	9
	10000	1328	9	40	—	15	93	—	—	—	4	5	12
	100	0	0	1	0	1	2	58	—	—	0	1	3
	300	0	1	1	0	2	4	25754	—	—	0	1	4
MTBhard	1000	1	3	4	1	5	13	—	—	—	1	1	38
	3000	2	7	12	4	5	31	—	—	—	1	1	338
	10000	8	15	38	—	14	70	—	—	—	4	5	1076

Table 7.2. Computing times in milliseconds as averages over 200 instances (INTEL PENTIUM 4, 1.5 GHz).

(BKP) into (KP). But as mentioned in Section 7.3.1 the effect of cardinality bounds is weakened by the transformation. Finally Bouknap has a very stable behavior, solving most instances within a fraction of a second. The strongly correlated instances demand more computational time, but the effort increases linearly with the problem size n and the data range R.

Thus the computational experiments indicate that specialized approaches for (BKP) are more efficient than those based on a transformation to (KP), since tighter upper bounds can be derived, and specialized reductions can be applied as described in Section 7.2.1.

7.4 Approximation Algorithms

Based on the concepts of approximation introduced in Sections 2.5 and 2.6 the corresponding results for (BKP) will be briefly presented.

The straightforward algorithm B-Greedy as described in Section 7.1 shows the same unfavourable behaviour as its analogon for (KP) since the same "bad" example with two item types $w_1 = 1$ and $p_1 = 2$ and $w_2 = p_2 = M$ with $c = M$ can be applied choosing $b_1 = 1$ and b_2 arbitrarily. The chosen value of $b_1 = 1$ makes the problem identical to a (KP) and hence may be seen as unfair. However, even if b_1 is slightly increased, e.g. to 2 or 3, the relative performance guarantee for B-Greedy can be forced to be arbitrarily close to 0 by a suitable selection of M.

This flaw can be easily corrected by defining Ext-B-Greedy for (BKP) which is the analogous extension of Greedy for (KP) from Section 2.5. At the end of B-Greedy

we simply add a comparison to the best solution achieved by filling the knapsack with copies of a single item type.

$$z^{eG} := \max\left\{z^G, \max_{j=1,\dots,n}\{b_j p_j\}\right\} \tag{7.22}$$

Recall assumption (7.4) which guarantees that b_j copies of any item type j will always fit into the knapsack.

Lemma 7.4.1 *Algorithm* Ext-B-Greedy *for (BKP) has a relative performance guarantee of* $\frac{1}{2}$ *and this bound is tight.*

Proof. Going through the proof of Theorem 2.5.4 it can be seen that the arguments used there apply even more so for (BKP). There is need to repeat the straightforward construction.

The tightness of the bound can be shown by almost the same example as for Theorem 2.5.4. Here we only need two item types with $w_1 = 1$, $p_1 = 2$ and $b_1 = 1$ for item type 1 and $w_2 = p_2 = M$ with $b_2 = 2$ for item type 2. Since $c = 2M$ the same argument as for (KP) applies. □

By definition of algorithm B-Greedy the exact analogon of Corollary 2.2.3 also holds for (BKP).

For the construction of a polynomial time approximation scheme (*PTAS*) for (BKP) two possibilities immediately come to mind. It was suggested in [335] and [394] to use the transformation from Section 7.1.1, which transforms (BKP) into an equivalent instance of (KP) with $O(n \log c)$ items, and to run any known *PTAS* for the resulting (KP). Although the number of items is polynomial, this approach still depends on the size of the input values which may be inconvenient for problems with a moderate number of items but huge coefficients. Note that the algorithm resulting from the application of H^ε (see Section 2.6) to the transformed instance of (KP) would have a running time of $O((n \log c)^\ell)$ with $\ell := \min\{\lceil \frac{1}{\varepsilon} \rceil - 2, n\}$.

A second strategy borrows only the main idea of the *PTAS* given in Section 2.6 for (KP). Recall that Algorithm H^ε basically tries to "guess" the ℓ items with largest profits in the optimal solution and fills the remaining capacity by the greedy algorithm. If the set of ℓ items has a high profit, then the missing part of the optimal solution cannot contribute too much to the optimal solution value. If this set has a relatively small profit, then all other items in the optimal solution set must have even smaller profits. Since the error induced by the greedy algorithm is bounded by the largest profit value of any item (see Corollary 2.2.3), this error cannot be too large.

To adapt this idea to (BKP) we also have to guess the ℓ items with highest profit. In contrary to (KP) there are now $\binom{n\ell}{\ell}$ possibilities to select these ℓ values, since every item type may contribute up to ℓ copies. Clearly, we have to take care to consider at

most b_j copies of every item type j. This yields a running time of roughly $O((n\ell)^\ell)$ without any technical improvements. The choice of ℓ is the same as above. Note that this expression depends only on n and $\frac{1}{\varepsilon}$.

Moving on in the "hierarchy" of approximation we consider the construction of an *FPTAS* for (BKP). Following the line of thought given in Martello and Toth [335] and in [394], namely the use of the transformation into an equivalent (KP), yields again an *FPTAS* with a running time still depending on the size of the input values. It would be preferable to have an algorithm whose time complexity depends only on n and $1/\varepsilon$. This goal can be reached if the (KP) resulting from the transformation is approximated by an *FPTAS* which has a running time not depending on the number of items n thus excluding the influence of $\log c$. This is not the case for most of the classical variants of an *FPTAS*, e.g. the one given in Section 2.6. However, some of the more sophisticated variants of *FPTAS* fulfill this property, e.g. algorithm FPKP by Kellerer and Pferschy [267, 266] described in Section 6.2. Note that n does appear in the complexity given in that algorithm but only because the greedy algorithm is applied to the "small" items. This part of FPKP could be easily replaced by running the corresponding greedy algorithm for the original (BKP).

To give a clearer picture of the resulting approximation scheme we will go into more details on this topic. This seems to be worthwhile because to our knowledge no explicit *FPTAS* for (BKP) was presented in the literature before.

Basically, we follow the same concept employed in many versions of an *FPTAS* for (KP). For the sake of readability the notation will be sometimes sloppy concerning the integrality of the expressions.

The approach consists of two parts. In the first part we partition the set of item types into *large* and *small* items. To do so, a lower bound z^ℓ on the objective function value is computed, e.g. by running Ext-B-Greedy and setting $z^\ell := z^{eG}$. Recall from Lemma 7.4.1 that $z^* \leq 2z^\ell$. All item types with $p_j \leq z^\ell \varepsilon$ are small item types. These are set aside while the main computation is performed for the set of large item types L with $p_j > z^\ell \varepsilon$.

This partitioning is necessary because a straightforward application of the scaling procedure given in Section 2.6 yielding the *FPTAS* referred to in Theorem 2.6.6 fails to work for our purposes. Note that the construction would require a polynomial bound on the number of items in the optimal solution set X^* which was trivially given by n for (KP) but which would introduce again a factor of $\log c$ for (BKP). Considering only the large item types guarantees that any feasible solution contains at most $\frac{2}{\varepsilon}$ copies of large item types because otherwise the total value of such a solution would exceed z^*.

In the main step of the first part of the algorithm, scaled dynamic programming by profits is applied to the large item types. This can be done by the corresponding algorithm given in Section 7.2. Recall that its running time was $O(\sum_{j=1}^{n} b_j U)$.

The scaling of the large item types is done in exactly the same way as for the items of (KP) in Section 2.6. This means that after partitioning the range of profits into intervals of the form $[iK,(i+1)K[$, every profit value is rounded down to the lower bound of its interval by setting $\tilde{p}_j := \lfloor \frac{p_j}{K} \rfloor$ for every item type $j \in L$.

After running dynamic programming by profits on the instance defined by the scaled large item types we get an optimal scaled solution vector $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_n)$. Evaluating the original profits for $\tilde{x}$ yields the approximate solution value z_L^A. Replicating the resulting series of inequalities (2.18) we get as a comparison to the optimal solution vector $\bar{x}^*$ for large item types only with solution value z_L^*

$$z_L^A = \sum_{j \in L} p_j \tilde{x}_j \geq \sum_{j \in L} K \left\lfloor \frac{p_j}{K} \right\rfloor \bar{x}_j^* \geq \sum_{j \in L} (p_j - K)\bar{x}_j^* = z_L^* - K \sum_{j \in L} \bar{x}_j^* .$$

In the second part of the algorithm we come back to the small item types. Obviously, the approximate solution may consist of an arbitrary combination of large and small item types. Of course, it may well happen that the small item types have a higher efficiency than the large ones and thus may contribute most of the optimal resp. approximate solution value. Therefore, we have to find out the best possible combination of profits contributed by the large and by the small item types. There does not seem to exist a more clever way to do that than going through all possible profit values generated by large item types from 0 up to $2z^\ell$ in the scaled profit space. This is done in the second part of the *FPTAS*.

Fortunately, the dynamic programming scheme computes during its execution not only the minimal capacity for the highest attainable profit value but at the same time the smallest capacities required for *all* profit values. Also the above inequality is valid in an analogous sense for all subproblems with smaller total profit.

Through the selection of K we want to guarantee that

$$\frac{z_L^* - z_L^A}{z_L^*} \leq \frac{K \sum_{j \in L} \bar{x}_j^*}{z_L^*} \leq \varepsilon \iff K \leq \frac{\varepsilon z_L^*}{\sum_{j \in L} \bar{x}_j^*} \tag{7.23}$$

holds (and hence is valid also for all subproblems). Proceeding along the same lines as in Section 2.6 we can calculate

$$\frac{\varepsilon z_L^*}{\sum_{j \in L} \bar{x}_j^*} \geq \frac{\varepsilon(z^\ell \varepsilon)}{2/\varepsilon} = \varepsilon^3 z^\ell / 2 .$$

Thus, choosing $K := \varepsilon^3 z^\ell / 2$ yields the desired approximation ratio fulfilling condition (7.23). By this choice every feasible solution represented in the scaled dynamic programming function differs by at most $z^\ell \varepsilon$ from its relative in an exact dynamic programming version.

It remains to be discussed how to combine large and small item types. As indicated above, this is done by going through all possible values of scaled profits contributed by the large item types. To be more precise, we consider every scaled profit value

$$0, \left\lfloor \frac{z^\ell \varepsilon}{K} \right\rfloor = \left\lfloor \frac{2}{\varepsilon^2} \right\rfloor, \left\lfloor \frac{2}{\varepsilon^2} \right\rfloor + 1, \ldots, \left\lfloor \frac{4}{\varepsilon^3} \right\rfloor = \left\lfloor \frac{2z^\ell}{K} \right\rfloor,$$

which requires roughly $\frac{4}{\varepsilon^3}$ iterations. For each candidate profit value we check the corresponding entry of the dynamic programming function. The capacity value given there is assumed to be consumed by large item types. Now the remaining free capacity of the knapsack is filled with copies of small item types by Ext-B-Greedy. Among all such candidates we naturally take the one which leads to the highest total profit and output the corresponding approximate solution.

Since the analogon of Corollary 2.2.3 holds also for (BKP) the error inflicted by running a greedy algorithm for the small item types instead of an exact algorithm is at most as large as the highest profit of a small item type. By definition this error of the second part of the algorithm is at most $z^\ell \varepsilon$.

Summarizing, we have constructed an approximate solution consisting of two parts each of which carrying an error of $z^\ell \varepsilon$ compared to the optimal solution value. Running the computation with an error bound of $\varepsilon/2$ yields the desired approximation ratio.

Concerning the running time of the dynamic programming part of the *FPTAS* described above the upper bound U in the running time complexity must be replaced by an upper bound on the scaled solution value. This is given by $2z^\ell/K = 4/\varepsilon^3$. Since the total number of large items in any feasible solution is bounded by $\frac{2}{\varepsilon}$, also the upper bounds b_j can never exceed this value. Hence, we get a running time of $O(n \cdot 1/\varepsilon^4)$ with space requirements of $O(n \cdot 1/\varepsilon^3)$ for the dynamic programming on the large item types in the first part.

The second part requires mainly the $O(1/\varepsilon^3)$ executions of the greedy algorithm each of them requiring $O(n \log n)$ time. This yields all together without any technical improvements a running time of $O(n \cdot 1/\varepsilon^4 + n \log n \cdot 1/\varepsilon^3)$. The space requirements of $O(n \cdot 1/\varepsilon^3)$ of the dynamic programming phase are sufficient for the whole algorithm.

It should be noted that this is only a rough estimate which could be improved by many of the techniques presented in the related sections referred to above. In particular, the partitioning of the profit values induced by the scaling could be done in a more diligent way. Taking into account that there are at most $\frac{2}{\varepsilon}$ items in any feasible solution, the width of the profit intervals depending on the value of K could be increased by setting $K := \varepsilon^2 z^\ell/2$ without forfeiting the required approximation ratio. Moreover, it is not necessary to start the greedy algorithm from scratch $1/\varepsilon^3$ times. It will be illustrated at the end of Section 8.5 how the best combination of small and large items can be found by one pass through the small items after sorting. However, the aim of this description was mainly to show how a straightforward *FPTAS* for (BKP) can be derived from simple building stones.

8. The Unbounded Knapsack Problem

The availability of an unlimited number of copies for every item type leads to phenomena which are quite different from the bounded case described in Chapter 7. Although there is a natural bound of how many copies of any item type can fit into a knapsack the structure of the problem is in several aspects not the same as for the case with a prespecified bound. Hence, it is worthwhile to devote this separate chapter to the unbounded knapsack problem (UKP).

8.1 Introduction

We repeat the precise definition of the problem from Section 1.2. A set of *item types* $N := \{1,\dots,n\}$ is given where all items of type j have profit p_j and weight w_j. There is an unlimited supply of identical copies of every item type j available. The corresponding integer programming formulation with optimal solution vector x^* and optimal solution value z^* is given as follows.

$$\text{(UKP)} \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{8.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c, \tag{8.2}$$

$$x_j \ge 0,\ x_j \text{ integer}, \quad j = 1,\dots,n. \tag{8.3}$$

Selecting an arbitrary number of copies from resources with unlimited but integer availability occurs in many practical decision scenarios. Applications are usually closely related to the case of (BKP) (see Section 7.1). A few selected scenarios will be included in Chapter 15, in particular in Section 15.2 and 15.4.

Although the range of the variables is unbounded, an instance of (UKP) with a *constant* number of variables can still be solved in polynomial time. This was shown for $n = 2$ by Hirschberg and Wong [230] and Kannan [257]. Note that this result is non trivial since a naive algorithm would have to go through all possible solutions

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

$x_1 = \ell$, $x_2 = \lfloor (c - \ell w_1)/w_2 \rfloor$ for $\ell = 0, \ldots, \lfloor c/w_1 \rfloor$, which is not polynomial in the input size.

An analogous result applies for general integer programs with $n = 2$ as shown by Scarf [418]. For any constant number n a polynomial running time algorithm for integer programming and hence also for (UKP) was given later in the famous paper by Lenstra [298].

The LP-relaxation of (UKP) is attained by replacing the integrality condition for x_j by $x_j \in \mathbb{R}^+$ for all $j = 1, \ldots, n$. As a straightforward extension of Corollary 7.1.1 or just as easily directly from Theorem 2.2.1 we get that the optimal solution vector of the LP-relaxation for (UKP) is defined by the item type with the largest efficiency p_j/w_j.

Lemma 8.1.1 *Let* $s := \operatorname{argmax}\{\frac{p_j}{w_j} \mid j = 1, \ldots, n\}$. *Then the optimal solution vector* $x^{LP} := (x_1^{LP}, \ldots, x_n^{LP})$ *of the LP-relaxation of (UKP) is given by*

$$x_s^{LP} := \frac{c}{w_s},$$
$$x_j^{LP} := 0 \quad \text{for } j \neq s,\ j = 1, \ldots, n.$$

Adapting algorithm Greedy of Section 2.1 can be done almost in the same way as for (BKP). In particular, the packing of a single item into the knapsack is replaced by packing as many copies as possible from a given item type constrained only by the available free capacity. There is only one trivial difference to the corresponding algorithm B-Greedy for (BKP) from Section 7.1. In algorithm U-Greedy for (UKP) the command for the assignment of x_j in Figure 7.1 is simplified from $x_j := \min\{b_j, \lfloor (c - \overline{w})/w_j \rfloor\}$ to $x_j := \lfloor (c - \overline{w})/w_j \rfloor$. In Section 8.5 we will comment on the performance of U-Greedy.

A transformation of (UKP) into an equivalent instance of (BKP) is easily attained by introducing the trivial bound

$$b_j := \left\lfloor \frac{c}{w_j} \right\rfloor \tag{8.4}$$

for every item type $j = 1, \ldots, n$. The resulting instance of (BKP) can be further transformed into a knapsack problem (KP) by the construction of Section 7.1.1 with the same complexity bound $O(n \log c)$ on the number of items. However, it is not very efficient in practice to tackle (UKP) by solving or approximating the corresponding instance of (KP). Consider that the drawbacks discussed in Section 7.1.1 apply even more so since the introduced bound b_j is of order $O(c)$ and fulfills the described "worst-case scenario" of very large coefficients.

A different kind of transformation concerns the *aggregation* of the objective function and of the constraint of (UKP). Indeed, it is possible to formulate (UKP) as

a "subset sum type" problem with identical coefficients in the objective and in the constraint. Note that such an aggregation is possible also for general binary linear optimization problems as discussed in Appendix A.2. For (UKP), this feature was described by Greenberg [194] based on much older results by Mathews [338]. Define $t := \lfloor p_s c/w_s \rfloor + 1$ and introduce an integer variable y indicating the "slack" in the constraint, i.e. $\sum_{j=1}^{n} w_j x_j + y = c$. Parameter t is not unlike a Lagrangian multiplier in some sense. Now consider the following problem.

$$\text{(UKP)}_A \quad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j + t\left(\sum_{j=1}^{n} w_j x_j + y - c\right) \tag{8.5}$$

$$\text{subject to} \quad \sum_{j=1}^{n} p_j x_j + t\left(\sum_{j=1}^{n} w_j x_j + y - c\right) \leq t - 1, \tag{8.6}$$

$$x_j \geq 0,\ x_j \text{ integer}, \quad j = 1,\ldots,n,$$

$$y \geq 0,\ y \text{ integer}.$$

The equivalence $(\text{UKP})_A$ and (UKP) was shown in [194] and [338]. We will give a simplified proof.

Theorem 8.1.2 *Problems (UKP) and $(\text{UKP})_A$ have the same optimal solution value z^* and $(x_1^*,\ldots,x_n^*)$ is part of the optimal solution vector of $(\text{UKP})_A$.*

Proof. Obviously, the optimal solution value of (UKP) fulfills

$$z^* \leq \lfloor z^{LP} \rfloor = \lfloor p_s c/w_s \rfloor = t - 1. \tag{8.7}$$

Defining the nonnegative slack in (8.2) of the optimal solution of (UKP) as $y^* := c - \sum_{j=1}^{n} w_j x_j^*$, we plug in x^* into the left-hand side of (8.6) and get with (8.7)

$$\sum_{j=1}^{n} p_j x_j^* + t\left(\sum_{j=1}^{n} w_j x_j^* + y^* - c\right) = z^* \leq t - 1. \tag{8.8}$$

This means that x^* is a feasible solution of $(\text{UKP})_A$.

Let $\bar{z}^*$ resp. $(\bar{x}^*,\bar{y}^*)$ be an optimal solution value resp. optimal solution vector of $(\text{UKP})_A$. Comparing (8.5) evaluated at x^* with the optimal solution value of $(\text{UKP})_A$ yields by performing the same evaluation as in (8.8)

$$z^* \leq \bar{z}^*. \tag{8.9}$$

Now we evaluate (8.5) for $\bar{x}^*$ and distinguish two cases depending in the sign of the last factor.

Case I: $\sum_{j=1}^{n} w_j \bar{x}^* + \bar{y}^* - c \geq 1$

Note that (8.6) can be written as $\bar{z}^* \leq t - 1$. Expressing t explicitly from (8.5) we get the contradiction

$$t = \frac{\bar{z}^* - \sum_{j=1}^{n} p_j \bar{x}_j^*}{\sum_{j=1}^{n} w_j \bar{x}_j^* + \bar{y}^* - c} \leq \bar{z}^* - \sum_{j=1}^{n} p_j \bar{x}_j^* \leq \bar{z}^* \leq t - 1.$$

Case II: $\sum_{j=1}^{n} w_j \bar{x}^* + \bar{y}^* - c \leq 0$

In this case $\bar{x}^*$ is clearly feasible for (UKP). Hence, we get from the condition of Case II

$$\bar{z}^* \leq \sum_{j=1}^{n} p_j \bar{x}_j^* \leq z^*.$$

Together with (8.9) this means that $z^* = \bar{z}^*$. Since x^* and $\bar{x}^*$ are both feasible for (UKP) and $(\text{UKP})_A$, the theorem is shown. □

In general, the aggregation of constraints results in rapidly increasing coefficients which makes the solution of the resulting instances of the subset sum problem impractical. A special algorithm for the solution of $(\text{UKP})_A$ which exploits its special structure was presented by Greenberg [194]. Babayev, Glover and Ryan [16] gave a more complicated aggregation framework together with a corresponding solution procedure. The resulting algorithm is based on group theory (cf. Glover [180]) and was reported to be more efficient than classical branch-and-bound methods.

8.2 Periodicity and Dominance

The unbounded knapsack problem has special structural properties which deserve a detailed discussion. Quite early in the history of research on knapsack problems Gilmore and Gomory [177] investigated the structure of the optimal solution vector of (UKP) and detected a certain *periodicity* for increasing capacity values.

A second structural property appears if item types are compared pairwise. If there exist two item types j and i with $p_j \geq p_i$ and $w_j \leq w_i$, it will never occur that an item of type i is part of the solution set since it would always be better (or at least not worse) to replace it by a copy of type j. Hence, item type i can be removed from consideration and a smaller problem remains to be considered. This feature of item type j *dominating* type i can be further extended and exploited for algorithmic purposes. Note that the basic definition of dominance was already introduced for (KP) in Section 3.4.

In this section we will take a closer look at these two interesting aspects. Both of them do not occur for (BKP) or for (KP) since in both of these cases the number of available items of any type is bounded. Thus it is not possible to use an item type periodically or as often as necessary to dominate all available copies of another item type.

8.2.1 Periodicity

The content of this section is based in the observation that the item type with highest efficiency, i.e. the item type s with maximal efficiency p_j/w_j (see Lemma 8.1.1), will contribute most of the profit to the solution value among all item types if the capacity is very large. Since there is an unlimited amount of copies of this item type available, it will be a good idea to pack many of them.

More formally, consider a packing with x_j copies of item type j. If there exist positive integers $a_j \leq x_j$ and b_j with $a_j w_j = b_j w_s$ then we can always construct a packing with higher profit by replacing a_j copies of item type j by b_j copies of item type s. This exchange operation will keep the weight of the packing unchanged but increases the overall profit by

$$b_j p_s - a_j p_j = \frac{a_j w_j}{w_s} p_s - a_j p_j = a_j w_j \left(\frac{p_s}{w_s} - \frac{p_j}{w_j} \right) \geq 0,$$

because of the definition of s. To derive a statement for the optimal solution vector let us refine the above argument and compute the *least common multiple* $\operatorname{lcm}(w_j, w_s) = a_j w_j = b_j w_s$ for every item type $j \neq s$. It follows that in every optimal solution vector there must be $x_j^* < a_j$ for $j \neq s$ independently from the given capacity c. Let $z^*(c)$ denote the optimal solution value of an unbounded knapsack problem with capacity c. Applying the same argument for all item types we get the following periodicity argument which was derived in a slightly more general and much more complicated version by Gilmore and Gomory [177]. As before $s = \arg\max \left\{ \frac{p_j}{w_j} \middle| j = 1, \ldots, n \right\}$ denotes the most efficient item type.

Theorem 8.2.1 *For every set of item types* $\{1, \ldots, n\}$ *let* $\operatorname{lcm}(w_j, w_s) = a_j w_j = b_j w_s$. *Setting*

$$c_0 := \sum_{\substack{j=1 \\ j \neq s}}^{n} a_j w_j, \tag{8.10}$$

there is for all $c \geq c_0$

$$z^*(c) = z^*(c - w_s) + p_s.$$

Proof. Consider an optimal solution vector x^*. If a copy of item type s is included in the packing, i.e. $x_s^* > 0$, then the statement trivially holds. Assume otherwise that $x_s^* = 0$. Since we assume $c \geq c_0$, there must be at least one item type $j \in \{1, \ldots, n\}$, $j \neq s$ which contributes at least a_j copies to the packing, i.e. $x_j^* \geq a_j$. However, it was shown above that $x_j^* < a_j$ holds for every $j \neq s$. This contradiction implies $x_s^* > 0$ and the theorem holds. □

Applying this theorem iteratively until the remaining capacity value falls below c_0 we get the following characterization of the optimal solution.

Corollary 8.2.1. *For all $c \geq c_0$, where c_0 is given by (8.10), there is*

$$z^*(c) = \left\lceil \frac{c - c_0}{w_s} \right\rceil p_s + z^* \left(c - \left\lceil \frac{c - c_0}{w_s} \right\rceil w_s \right).$$

This means that in order to solve an instance of (UKP) with arbitrary capacity c it is sufficient to compute the optimal solution for all capacities smaller than c_0 independently from c and add an appropriate number of copies of item type s. It was observed in practical experiments that the actual value of c_0, where the periodicity starts to occur, is usually much smaller than the value given in (8.10).

A practical algorithm to solve (UKP) by computing the periodic solution was given by Greenberg [193]. Note that also the construction of the aggregated problem $(\text{UKP})_A$ is closely connected to this periodicity property. Further algorithms exploiting the periodicity of the optimal solution value in some way or other will be mentioned in Section 8.3.

8.2.2 Dominance

Simple dominance between two item types as described above can be considerably extended. The earliest treatment of dominance is due to Gilmore and Gomory [175]. The concept was extended by Martello and Toth [334] to the so-called *multiple dominance*. Although several other authors contributed to the topic (see e.g. Babayev and Mardanov [17], Dudzinski [114], Zhu and Broughan [500]), the most general and with respect to algorithmic applicability the most efficient notion of dominance, namely *threshold dominance*, was recently presented by Andonov et al. [11]. Since their approach covers all previously introduced versions of dominance we will restrict ourselves to their model.

The thorough treatment of the advanced domination concepts requires a number of more formal definitions and notations in increasing order of generality.

Definition 8.2.2

1. *An item type j* simply dominates *item type i, $i \neq j$, written as $j \gg_s i$, if*

$$p_j \geq p_i \text{ and } w_j \leq w_i.$$

2. *An item type j* multiply dominates *item type i, $i \neq j$, written as $j \gg_m i$, if*

$$\left\lfloor \frac{w_i}{w_j} \right\rfloor \geq \frac{p_i}{p_j}.$$

3. *A set of item types $I \subseteq N$* collectively dominates *item type i, $i \notin I$, written as $I \gg_c i$, if there exists a vector $y = (y_1, \ldots, y_n)$, $y_j \in \mathbb{N}_0$, such that*

$$\sum_{j \in I} y_j w_j \leq w_i \text{ and } \sum_{j \in I} y_j p_j \geq p_i.$$

4. *A set of item types $I \subseteq N$* threshold dominates *item type i, $i \notin I$, written as $I \gg_t i$, if there exists a multiplier $\alpha \in \mathbb{N}$ and a vector $y = (y_1, \ldots, y_n)$, $y_j \in \mathbb{N}_0$, such that*

$$\sum_{j \in I} y_j w_j \leq \alpha w_i \text{ and } \sum_{j \in I} y_j p_j \geq \alpha p_i.$$

Clearly, $j \gg_s i$ means that in every feasible packing every copy of item type i can be replaced by a copy of item type j without increasing the total weight and without decreasing the total profit of the packing. Graphically, every item type and every solution set can be seen as a point in a (weight, profit) space. Dominance corresponds to the dominating point lying in the upper left quadrant seen from the dominated point. The other way round, any reachable (weight, profit) point dominates all items lying in the lower right corner of this point. A graphical representation of all four cases of dominance in a (weight, profit) space is given in Figure 8.1.

Note that the definition of multiple dominance implies $\lfloor w_i/w_j \rfloor \geq 1$. Furthermore we have $\left\lfloor \frac{w_i}{w_j} \right\rfloor p_j \geq p_i$ and $\left\lfloor \frac{w_i}{w_j} \right\rfloor w_j \leq w_i$. Hence, multiple dominance $j \gg_m i$ indicates that it is always better to pack $\lfloor w_i/w_j \rfloor$ copies of item type j rather than a single item of type i as depicted in Figure 8.1.

Algorithms to find all multiply dominated item types were developed e.g. by Martello and Toth [334] with a running time of $O(n^2)$, Dudzinski [114] with an improved $O(n \log n + mn)$ time algorithm where $m \leq n$ is the number of undominated item types. Moreover, Pisinger [380] improved the running time to $O(n \log n + \min\{mn, n w_{\max}/w_{\min}\})$. A statistical analysis underlining the small number of undominated item types which may be expected for randomly generated instances of (UKP) was performed by Johnston and Khan [255].

The more complicated case of collective dominance $I \gg_c i$ says that there exists an integer linear combination of item types which together is more efficient than item type i. The correctness of this assertion is obvious from the definition.

Finally, threshold dominance $I \gg_t i$ is a generalization of collective dominance, where an integer linear combination of item types together is more efficient than a certain multiple α of item type i. Clearly, $\alpha = 1$ is equivalent to collective dominance. These two cases are illustrated in the lower half of Figure 8.1.

The considerations about the role of dominance in the optimal solution yield the following proposition.

Proposition 8.2.3 *For every instance of (UKP) there always exists an optimal solution* **not** *containing any simply, multiply or collectively dominated item types.*

Moreover, upper bounds can be introduced for the number of copies of threshold dominated item types.

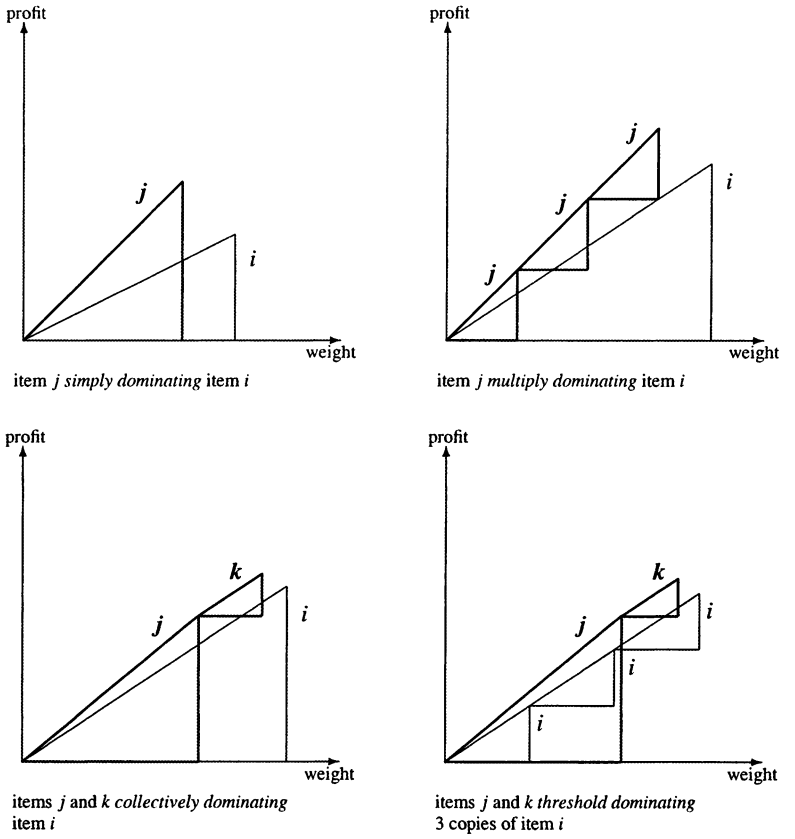


Fig. 8.1. The four cases of dominance for item i: Thick lines indicate the profit and weight structure of the dominating item types, thin lines correspond to a dominated item type.

The algorithmic application of the introduced dominance concepts requires an appropriately adapted framework. An advanced algorithm incorporating dominance in a very efficient way will be given in detail in Section 8.3.2.

Several authors dealt with the identification of special cases of (UKP) where certain dominance relations are given such that a simple polynomial algorithm or even a greedy approach solves the problem to optimality. The corresponding situation for (KP) was briefly discussed at the beginning of Chapter 5.

In particular, the minimization version of (UKP), where the objective function should be as small as possible under the constraint $\sum_{j=1}^{n} w_j x_j \geq c$, received attention e.g. by Magazine, Nemhauser and Trotter [313], Hu and Lenhard [239], Tien and Hu [458] and Zukerman et al. [503]. In the latter paper the following conditions are presented for a minimization (UKP):

$$w_j < w_{j+1} \text{ and } p_j < p_{j+1}, \text{ for } j = 1,\ldots,n-1, \tag{8.11}$$

$$p_{j+1} \leq \lfloor w_{j+1}/w_j \rfloor p_j, \text{ for } j = 1,\ldots,n-1. \tag{8.12}$$

Note that (8.11) excludes only simply dominated item types in analogy to the maximization version and can be assumed without loss of generality after reordering the item types. Condition (8.12) corresponds to item type $j+1$ multiply dominating item type j (in the minimization sense), thus item type n would be the most attractive item. In the minimization case, the resulting optimal solution is not obvious but can be computed by a recursive algorithm in $O(n)$ time. Directly analogous conditions for the maximization case are not meaningful, since the item type with minimal weight would multiply dominate all other item types and constitute a trivial optimal solution.

8.3 Dynamic Programming

Naturally, the basic dynamic programming approach for (BKP) as described and improved in Section 7.2 can be applied immediately to (UKP) by introducing the trivial bound (8.4). However, the fact that we do not have to worry about the number of copies we pack from every item type, calls for an explicit statement of the resulting recursion.

Let the optimal solution value of an instance of (UKP) restricted to the item types $1,\ldots,j$, $0 \leq j \leq n$, and capacity d with $0 \leq d \leq c$ be denoted by

$$z_j(d) := \max\left\{ \sum_{\ell=1}^{j} p_\ell x_\ell \;\middle|\; \sum_{\ell=1}^{j} w_\ell x_\ell \leq d, x_\ell \in \mathbb{N}_0, \ell = 1,\ldots,j \right\}. \tag{8.13}$$

As an extension of recursion (2.8) from Section 2.3 we can formulate similar to (7.7) for (BKP)

$$z_j(d) := \max_{\ell}\{z_{j-1}(d-\ell w_j) + \ell p_j \mid d \geq \ell w_j, \ell \in \mathbb{N}_0\}. \tag{8.14}$$

Parameter ℓ denotes the number of copies of item type j to be included into a knapsack with capacity d. Different from (7.7), in the case of (UKP) the computation of $z_j(d)$ can also rely on the optimal solution value of subproblems on the *same set of item types* but smaller capacities. Therefore, the recursion can also be evaluated as

$$z_j(d) := \begin{cases} z_{j-1}(d) & \text{if } d < w_j, \\ \max\{z_{j-1}(d), z_j(d-w_j) + p_j\} & \text{if } d \geq w_j, \end{cases} \tag{8.15}$$

for $d = 1,\ldots,c$ and $j = 1,\ldots,n$ after initializing $z_j(0) = 0$ for $j = 0,\ldots,n$, and $z_0(d) = 0$ for $d = 1,\ldots,c$. Computing $z_j(d)$ by (8.15) will give way to a straightforward dynamic programming algorithm Unbounded-DP (see below) which runs in $O(nc)$ time and requires only $O(n+c)$ space, even without applying the storage reduction technique of Section 3.3.

8.3.1 Some Basic Algorithms

The following implementation of Unbounded-DP, as depicted in Figure 8.2, can be seen as a variant of DP-3 from Section 2.3. However, we implicitly include the possibility to add an arbitrary number of copies of every item type j. This is achieved by performing the update operations not in decreasing order of weights, i.e. for $d := c$ downto w_j, where the update candidates are compared only with entries $z(d - w_j)$ which were not yet updated by item j. Instead we follow the definition of (8.15) and go from w_j up to c thereby comparing the update candidate with entries $z(d - w_j)$, which were already candidates for a possible update by copies of item type j. Recall that this would be impossible for (KP) or (BKP) since we loose any control of how many times an item is included in a subset corresponding to a particular dynamic programming function entry.

Algorithm Unbounded-DP:

```
for d := 0 to c do
    z(d) := 0, r(d) := 0        initialization
for j := 1 to n do
    for d := w_j to c do        item of type j may be packed
        if z(d − w_j) + p_j > z(d) then
            z(d) := z(d − w_j) + p_j
            r(d) := j
z* := z(c)
X* := ∅, c̄ := c
repeat        recovering the optimal solution set
    r := r(c̄)
    X* := X* ∪ {r}
    c̄ := c̄ − w_r
until c̄ = 0
```

Fig. 8.2. Basic dynamic programming algorithm Unbounded-DP for (UKP) including the computation of X^*.

The second relevant difference is the fact the we can recover the optimal solution vector by accessing only the final array $r(d)$. This was impossible in DP-3 since after extracting the first item from $r(c)$ it may well happen that the same item appears again in the subset corresponding to the optimal solution for capacity $c - w_{r(c)}$ or this double occurrence of an item may appear later in the process. Clearly, this is not a matter of concern for (UKP) which simplifies the reconstruction of X^* considerably.

Obviously, this algorithm also solves the all-capacities version of the problem. Since similar or related constructions can be found in many textbooks, we refrain from pointing out any particular reference.

Theorem 8.3.1 *Algorithm* Unbounded-DP *solves (UKP) in $O(nc)$ running time and $O(n+c)$ space.*

Proof. The correctness of Unbounded-DP was described above. The claimed complexities follow immediately from the description of the algorithm. □

It was noted in several textbooks (e.g. Wolsey [489, ch.5], Nemhauser and Wolsey [360, ch.II.3.4]) that the solution of (UKP) can also be modeled as a *longest path problem* in an acyclic, weighted, directed graph $G = (V,A)$. This observation was elaborated by Shapiro [431] in 1968 but may have been known before. We presume elementary knowledge of algorithmic graph theory which can be found e.g. in Jungnickel [256].

The graph is given by a vertex set $V := \{0,\ldots,c\}$ and two types of arcs. For every item type $j \in N$ there are arcs $(v, v+w_j)$ for $v \in V$ and $v+w_j \leq c$ with length p_j. Furthermore, there are arcs (v,c) for $v = 0,\ldots,c-1$ with length 0. Every vertex v stands for a knapsack with total weight v. Moving from vertex v along an arc $(v, v+w_j)$ illustrates that a copy of item type j is added to the knapsack. Every path from vertex 0 to vertex v, $v \in V$, corresponds to a feasible packing of the knapsack. The profit of such a packing is given by the sum of arc lengths along the path. We will illustrate this graph representation in Figure 8.3. based on the following example.

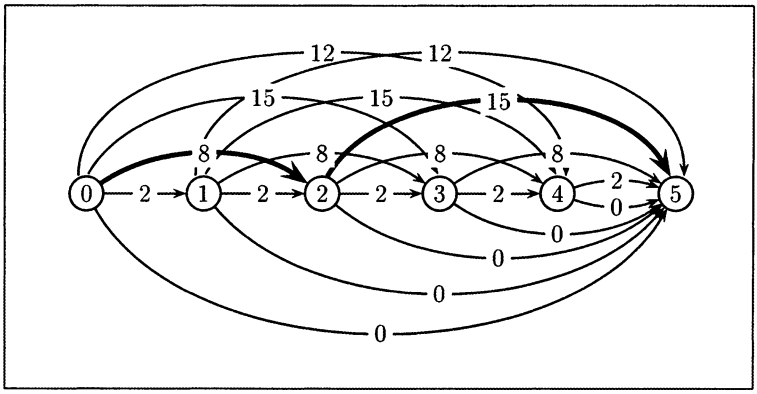


Fig. 8.3. Representation of an instance of (UKP) with 4 item types and capacity 5 as a directed graph. The optimal solution is given by the path from 0 to 5 drawn in boldface.

Example: Consider an instance of (UKP) with capacity $c = 5$ and $n = 4$ item types with the following data:

j	1	2	3	4
p_j	2	8	15	12
w_j	1	2	3	4

The full directed graph G corresponding to this example is depicted in Figure 8.3. Every path from vertex 0 to vertex 5 corresponds to a feasible solution of the given instance. The optimal solution value 23 is derived by packing one copy of item type 2 and item type 3 as indicated by the two boldface arcs. □

It can be easily seen that finding the optimal solution of (UKP) is equivalent to the computation of a longest path in this graph from vertex 0 to vertex c. The arcs with length 0 are necessary to include solutions which do not completely fill the knapsack capacity. This construction only works for (UKP) but not for (KP) or (BKP) since there is no way of controlling how many copies of each item type are packed.

For later use in the design of approximation algorithms let us briefly mention dynamic programming by profits. As before, $y_j(q)$ denotes the minimal weight of a collection of copies from item types $1,\dots,j$ with total profit q. Basically, the same approach as given in Section 7.2 can be applied with (7.8) simply replaced by

$$y_j(q) := \min_{\ell}\{y_{j-1}(q-\ell p_j)+\ell w_j \mid q \geq \ell p_j, \ell \in \mathbb{N}_0\}. \tag{8.16}$$

The above algorithm Unbounded-DP can be easily converted into dynamic programming by profits by exchanging the roles of profits and weights after computing an upper bound U on the optimal solution value, e.g. by a greedy-type algorithm. The convenient handling of the optimal solution vector remains unchanged. In the running time and space complexity the capacity c is replaced by U.

More advanced and specialized dynamic programming algorithms for (UKP) were proposed by several authors. We will give a brief description of the interesting strategy of the algorithm by Greenberg and Feldman [196] and then concentrate on the algorithm by Andonov, Poirriez and Rajopadhye [11], which was shown to be very efficient in practical experiments.

If the capacity c of (UKP) is very large compared to the values of the item weights, the statement of Corollary 8.2.1 can be exploited algorithmically. Consider that in such a situation the main part of the knapsack is filled with copies of item type s (indicating the item type with the highest efficiency). Hence, it is not efficient to compute an optimal packing for all capacity values $d = 1,\dots,c$ to find out how many copies of item s are actually packed. The following approach due to Greenberg and Feldman [196] introduces a transformation of (UKP) leading to the computation of the capacity *not filled* by items of type s.

Obviously, there can be at most $\lfloor c/w_s \rfloor$ copies of item type s in any feasible solution. Let us denote by y_s the number of copies of type s which are *not packed*. After determining the optimal value of y_s, we clearly have $x_s^* = \lfloor c/w_s \rfloor - y_s$. In the light of Corollary 8.2.1 it can be expected that the optimal value of y_s is relatively small for very large c. To find this value we introduce the following integer program which is equivalent to (UKP).

$$\text{maximize} \quad \sum_{\substack{j=1 \\ j \neq s}}^{n} p_j x_j + p_s \lfloor c/w_s \rfloor - p_s y_s \tag{8.17}$$

$$\text{subject to} \quad \sum_{\substack{j=1 \\ j \neq s}}^{n} w_j x_j \leq c - w_s \lfloor c/w_s \rfloor + w_s y_s ,$$

$$x_j \geq 0,\ x_j \text{ integer}, \quad j = 1, \ldots, n,$$

$$y_s \in \{0, \ldots, \lfloor c/w_s \rfloor\}.$$

Since the classical dynamic programming approaches solve (UKP) to optimality not only for capacity c but also for the all-capacities version, we can use any such algorithm to solve the above formulation of (UKP) by going through all right-hand sides of the constraint induced by some feasible value of y_s. The main point of this approach is the fact that we expect the optimal right-hand side to be quite small compared to the original capacity c. To gain from this feature, the order of the dynamic programming computation must be reversed. Instead of considering the items one after the other and checking all capacity values for each of them, we go through all capacities and try to improve the corresponding dynamic programming function entries by including one of the items.

Greenberg and Feldman [196] gave an upper bound on y_s limiting the maximal capacity value which needs to be considered. Since the running time of dynamic programming is dependent on the capacity (see Theorem 8.3.1), this approach can be expected to improve the overall performance.

8.3.2 An Advanced Algorithm

A rather sophisticated variant of dynamic programming taking into account the various notions of dominance was recently presented by Andonov, Poirriez and Rajopadhye [11], Their algorithm EDUK (*Efficient Dynamic programming for the Unbounded Knapsack problem*) seems to be the most efficient dynamic programming based method available at the moment. Its main feature is the detection of various notions of dominance not only as a preprocessing step but also during the dynamic programming computation. The main point of the latter is to eliminate item types from consideration, after detecting that they are dominated, also during the iterations.

However, this aspect cannot be realized by the dynamic programming recursion as given in Section 2.3. Note that the computation in (2.8) processes one item type completely (i.e. for all capacity values) and then never comes back to it after moving on to the next item type. Hence, the elimination of an item type which was already processed before does not yield any savings whereas domination of future item types is impossible to predict.

Therefore, the dynamic programming function $z_j(d)$ defined in (8.14) has to be evaluated in a different order, namely by computing its value completely for one *capacity value* (i.e. for all item types) before incrementing the capacity. In this way, we gain from detecting dominance of an item type at some capacity value, since this item type can be eliminated and not considered anymore for all higher capacity values.

It is beyond the scope of this book to give a full presentation of EDUK which includes many implementational features to improve its performance. We omit most of these technical details and concentrate on the main approach depicted in Figure 8.5. The interested reader is referred to the original paper [11] and the links provided there.

The main theoretical contribution in the design of EDUK is the test for collective dominance of an item type by a previously computed entry of the dynamic programming array. Clearly, these optimal solutions of subproblems are in some sense the best available candidates to collectively dominate an item type and thus give the highest chances to reduce the number of iterations. Of course, the set of candidates for collective dominance is restricted to the item types previously processed by dynamic programming.

The entries of the dynamic programming function computed by (8.14) can be seen as a $(c \times n)$ table with one row for every capacity value and one column for every item type. This table is processed row by row always taking several rows together as a *slice*. Whenever an item type is found to be dominated, the corresponding column can be deleted. Hence, the evaluation is performed only for the set of currently undominated (free) columns F. Moreover, all item types j with $w_j > d$ do not have to be considered for a row with capacity d.

In principle, EDUK consists of two phases. In the first phase (8.14) is evaluated for all rows with capacity d not larger than the highest weight $w_{\max}$ of an item type. During this phase we perform iterations in each of which a set of t item types is considered. In particular, after sorting the item types in increasing order of weights, in an iteration k a subset I of the item types $(k-1)t+1,\ldots,kt$ is generated excluding any dominated types. This results in a slice of variable size containing all capacities between $w_{(k-1)t}+1$ and w_{kt}. We compute the best possible solution values for *all entries* in this slice but only for the currently undominated item types. If the current capacity d is equal to the weight of an item f in I, it is checked whether a single copy of f improves the current best solution value.

In the second phase the evaluation of (8.14) is continued for all capacities d larger than the highest item weight $w_{\max}$. This time we proceed in iterations covering slices of fixed size q. At the end of each slice evaluation the set of currently undominated item types F is updated. The partitioning of the dynamic programming table into slices is illustrated in Figure 8.4.

The two parameters t and q defining the size of a slice in the two phases have to be chosen appropriately, usually through computational tests. Their magnitude is a

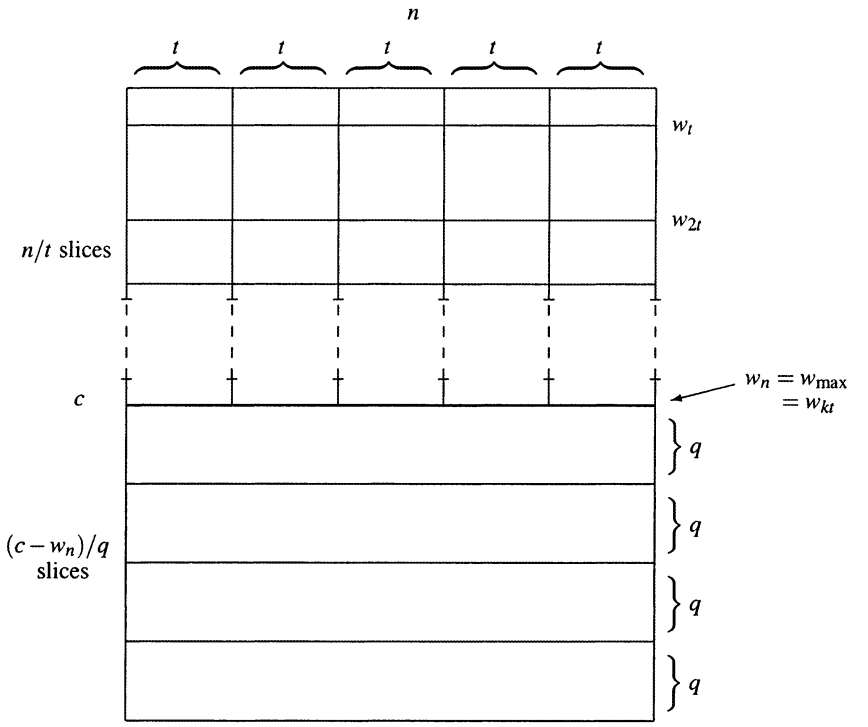


Fig. 8.4. Partitioning of the dynamic programming table by EDUK. The item types are sorted in increasing order of weights. The first phase processes the capacity values from 1 to $w_n = w_{\max}$ in slices each of which covers the values until the next t-largest item weight. The second phase goes through the remaining capacity values $w_n + 1$ until c in slices of fixed size q.

trade off between detecting dominance as early as possible and the computational effort to check dominance. A quite technical description of EDUK still omitting many details is given in Figure 8.5. For simplicity of notation it is assumed that n/t is integer.

After applying trivial dominance criteria as a preprocessing step to the set of item types, dominance according to Definition 8.2.2 is examined in three ways during the execution of EDUK.

Dominance 1 reduces the set of item types $\{(k-1)t+1,\ldots,kt\}$ to the subset I by checking collective dominance of every item type by the currently best solution set for capacity $w_{(k-1)t}$. Moreover, it is also checked whether an item type is multiply dominated by the item type in F with the highest efficiency since this is the item type with the best chance to achieve dominance.

Sketch of Algorithm EDUK:

Initialization
sort the item types such that $w_1 \leq w_2 \leq \ldots \leq w_n$
$I := \emptyset$ *item types to be inserted in the current slice*
$F := \emptyset$ *undominated item types*
set $last_F := 0$ and $pred_F := 0$ for the first element of F
$k := 1$, $z(0) := 0$, $w_0 := 0$, $w_{n+1} := c+1$ *dummy item types*
Slice evaluation for capacity 1 to w_n
repeat *a subset of size t of all item types is considered*
  for $j := (k-1)t+1$ to kt do
    check **dominance 1** for item type j
    if j is not dominated then add j to I
  I contains a subset of item types $\{(k-1)t, \ldots, kt\}$
  $f := \arg\min\{w_i \mid i \in I \cup \{n+1\}\}$
  $g(0) := 0$ $g(j)$ *denotes* $z_j(d)$
  for $d := w_{(k-1)t}+1$ to w_{kt} do *evaluation of a slice*
    for all $j \in F$ do
      $g(j) := \max\{g(pred_F(j)), z(d-w_j)+p_j\}$
    if $d = w_f$ then
      $g(f) := \max\{g(last_F), p_f\}$
      check **dominance 2** for item type f
      if f is not dominated then add f to F
      remove f from I
      $f := \arg\min\{w_i \mid i \in I \cup \{n+1\}\}$
    $z(d) := \max\{g(j) \mid j \in F \cup \{0\}\}$
  end for d
  check **dominance 3** for F
  $k := k+1$
until $kt > n$
Slice evaluation for capacity w_n+1 **to** c
$\bar{c} := w_{(k-1)t}+1$
repeat *evaluation of the remaining slices of fixed size q*
  for $d := \bar{c}$ to $\bar{c}+q-1$ do
    for all $j \in F$ do $g(j) := \max\{g(pred_F(j)), z(d-w_j)+p_j\}$
    $z(d) := \max\{g(j) \mid j \in F\}$
  end for d
  check **dominance 3** for F
  $\bar{c} := \bar{c}+q$
until $\bar{c} > c$
$z^* := z(c)$

Fig. 8.5. Sketch of algorithm EDUK with parameters t (variable slice size) and q (fixed slice size for capacity larger than w_n) which solves (UKP) by dynamic programming exploiting different notions of dominance. F is organized as ordered list with highest element $last_F$ and $pred_F(j)$ retrieving the predecessor of $j \in F$. In order to reduce space consumption the table $g(j)$ is used to save $z_j(d)$.

When a new item type f is introduced in the evaluation, **dominance 2** checks whether f is collectively dominated by F. Finally, throughout the computation with fixed slices when all item types were already considered at some point, **dominance 3** tries to reduce F by checking for threshold dominance at the end of each slice. The corresponding criterion is given in [11].

Another main feature of EDUK, which we do not cover in detail, is the use of *sparse computation*. The main idea thereof is to exploit the monotonicity of $z_j(d)$ in d. Note that a column in the dynamic programming array usually does not contain a different profit value for every capacity value but consits of a number of intervals each of which containing entries with identical profit values. Clearly, the profits of these intervals are increasing with d. Therefore, some effort can be saved by computing and storing only the endpoints of the intervals containing identical values. For more details on this aspect see Andonov and Rajopadhye [12]. Finally, it should be pointed out that the worst-case running time of EDUK is still $O(nc)$.

8.3.3 Word RAM Algorithm

In the case of the *unbounded subset sum problem*, i.e. the case where $p_j = w_j$ for $j = 1,\ldots,n$, Pisinger [392] showed how to use word-parallelism as described in Section 3.7 to improve the time complexity. As usually we will assume that the word size W is of magnitude $\Theta(\log c)$.

The algorithm makes use of the same binary coding as described in Section 4.1.1, hence we have the table $g(j,\overline{w})$ for $j = 1,\ldots,n$ and $\overline{w} = 0,\ldots,c$ defined by

$$\begin{aligned} &g(j,\overline{w}) = 1 \Leftrightarrow \\ &\text{there is a subset of } \{w_1,\ldots,w_j\} \text{ with overall sum } \overline{w} \end{aligned} \tag{8.18}$$

Initially we set $g(0,0) = 1$ and $g(0,\overline{w}) = 0$ for $\overline{w} = 1,\ldots,c$. Subsequent values of g are derived as

$$g(j,\overline{w}) = 1 \quad \Leftrightarrow \quad \Big(g(j-1,\overline{w}) = 1 \quad \vee \quad g(j,\overline{w}-w_j) = 1\Big) \tag{8.19}$$

for $j = 1,\ldots,n$ and $\overline{w} = 0,\ldots,c$. Notice that the above recursion *must* be evaluated for increasing values of $\overline{w}$ to ensure that the weight w_j can be chosen an arbitrary number of times. By storing W (which is in $O(\log c)$) entries of table g in each word we get the improved space complexity $O(n + c/\log c)$. In order to also improve the time complexity we would like to use binary operations on words of size W to get from table $g(j-1,\cdot)$ to table $g(j,\cdot)$. However, this is complicated by the fact that recursion (8.19) contains references to $g(j,\cdot)$ and hence may refer to the same word. To work around this problem we distinguish between small weights with $w_j \leq W$ and large weights with $w_j > W$.

As we may delete duplicated weights w_j, at most W of the weights will be smaller than $\log c$, and we may consider them in increasing order. Running a revised recursion, where in the j-th step the word-length is chosen equal to w_j. Thus if $w_j = 1$ we get an "ordinary" recursion of the form

$$g(j,\overline{w}) = 1 \quad \Leftrightarrow \quad \begin{pmatrix} g(j-1,\overline{w}) = 1 & \vee \\ g(j,\overline{w}-w_j) = 1 & \vee \\ g(j-1,\overline{w}-w_j) = 1 & \end{pmatrix} \tag{8.20}$$

If $w_j = 2$, we consider two bits in each iteration, and so on. The time complexity of evaluating recursion (8.19) becomes less than

$$c + \frac{c}{2} + \frac{c}{3} + \frac{c}{4} + \cdots + \frac{c}{\lceil \log c \rceil} = c \sum_{j=1}^{\lceil \log c \rceil} \frac{1}{j} = c H_{\lceil \log c \rceil},$$

where the harmonic number H_n is of magnitude $\log n + O(1)$. This gives the total running time $O(c \log\log c)$ for the first part of the algorithm.

For the remaining weights with $w_j \geq \log c$ the following technique is used: Let $g(j-1,\cdot)$ be a binary string representing those sums which can be obtained with the integers $w_1,\ldots,w_{j-1}$. Take the lowest order word in the binary string, shift it to the right by w_j positions and or it to $g(j-1,\cdot)$. Take the second-lowest order word in $g(j-1,\cdot)$ and repeat the operation. Continue this way until all words in $g(j-1,\cdot)$ have been considered. This implies that the value w_j can be added to the sum any number of times, since it is reintroduced to the sum. Still the complexity of a single merging of binary strings takes $O(c/\log c)$.

In total, the time complexity becomes $O(c \log\log c + nc/\log c)$ which corresponds to $O(nc/\log c)$ for sufficiently large values of n.

8.4 Branch-and-Bound

The upper bounds required for a branch-and-bound algorithm can be derived basically along the same lines as for (KP) in Section 5.1.1 and for (BKP) in Section 7.3.1. Assuming the item types to be sorted according to decreasing efficiencies the rounded down solution of the LP-relaxation yields the trivial bound

$$U_1 := U_{\mathrm{LP}} = \lfloor z^{LP} \rfloor = \left\lfloor p_1 \frac{c}{w_1} \right\rfloor.$$

As a counterpart to (5.12) resp. (7.19) we can further consider the possibility to fill the residual capacity $c_r := c - w_1 \left\lfloor \frac{c}{w_1} \right\rfloor$ left over after packing the maximal number of copies of item type 1 with copies of item type 2.

The number of such copies which can always be packed is given by $\beta_2 := \lfloor c_r/w_2 \rfloor$. Then we distinguish between two cases. Either after packing $\lfloor c/w_1 \rfloor$ items of type 1 there are exactly β_2 copies of type 2 packed and the remaining capacity is filled by item type 3. Or there are $\beta_2 + 1$ copies of type 2 packed and as many items of type 1 as possible. This number is given by $\beta_1 := \left\lfloor \frac{c - w_2(\beta_2+1)}{w_1} \right\rfloor$. The capacity remaining afterwards is filled again by item type 2. This yields the following bound by Martello and Toth [334]:

$$U_2 := \max \left\{ \left\lfloor p_1 \left\lfloor \frac{c}{w_1} \right\rfloor + p_2\beta_2 + (c_r - w_2\beta_2)\frac{p_3}{w_3} \right\rfloor, \right. \tag{8.21}$$
$$\left. \left\lfloor p_2(\beta_2+1) + p_1\beta_1 + (c - w_2(\beta_2+1) - w_1\beta_1)\frac{p_2}{w_2} \right\rfloor \right\}.$$

Obviously, the maximum cardinality bounds introduced for (BKP) in (7.21) are not meaningful for (UKP) since they degenerate to the trivial bound $\sum_{j=1}^{n} x_j \leq \left\lfloor \frac{c}{w_{\min}} \right\rfloor$.

A rather unusual upper bound, which combines weights and profits in a non-trivial way, was recently proposed by Poirriez, Yanev and Andonov [396]. We will give a slightly modified presentation and a simplified proof of the construction. Let us assume that 1 is the item type with smallest weight, i.e. $w_1 = w_{\min}$. Then we define the scaling parameter $\psi := \left\lceil \frac{w_1+1}{p_1} \right\rceil$ with $\psi \geq 1$ and multiply all profit values by ψ which yields $p'_j := \psi p_j$ for $j = 1,\ldots,n$. The aim of this scaling is to guarantee $p'_1 - w_1 > 0$.

Now define for $j = 1,\ldots,n$ the values

$$\gamma_j := \left\lfloor \frac{w_j}{w_1} \right\rfloor \quad \text{and} \quad \delta_j := \frac{p'_j - w_j}{p'_1 - w_1}\frac{1}{\gamma_j}.$$

Moreover, let $\delta_{\max} := \max_{j=1,\ldots,n}\{\delta_j\}$. Since $\delta_1 = 1$, there is $\delta_{\max} \geq 1$. Then the following upper bound was given in [396].

Theorem 8.4.1

$$c + \left\lfloor \delta_{\max}(p'_1 - w_1) \left\lfloor \frac{c}{w_1} \right\rfloor \right\rfloor =: U_3 \geq z^*$$

Proof. The complicated definition of $\delta_{\max}$ is required to guarantee that for $j = 1,\ldots,n$ there is

$$p'_j - w_j \leq \delta_{\max}(p'_1 - w_1)\gamma_j, \tag{8.22}$$

which is trivial to check. Now the optimal solution value of (UKP) can be bounded by plugging in (8.22)

$$
\begin{aligned}
z^* &= \sum_{j=1}^{n} p_j x_j^* \le \sum_{j=1}^{n} p'_j x_j^* \le \sum_{j=1}^{n} p'_j x_j^* + c - \sum_{j=1}^{n} w_j x_j^* \\
&= \sum_{j=1}^{n} (p'_j - w_j) x_j^* + c \\
&\le c + \sum_{j=1}^{n} \delta_{\max}(p'_1 - w_1)\gamma_j x_j^* \\
&\le c + \delta_{\max}(p'_1 - w_1) \sum_{j=1}^{n} \left\lfloor \frac{w_j}{w_1} \right\rfloor x_j^* .
\end{aligned}
$$

Applying basic properties of the floor function we conclude the proof with

$$
\sum_{j=1}^{n} \left\lfloor \frac{w_j}{w_1} \right\rfloor x_j^* \le \sum_{j=1}^{n} \left\lfloor \frac{w_j x_j^*}{w_1} \right\rfloor \le \left\lfloor \sum_{j=1}^{n} \frac{w_j x_j^*}{w_1} \right\rfloor \le \left\lfloor \frac{c}{w_1} \right\rfloor .
$$

□

Poirriez, Yanev and Andonov [396] showed that for $\delta_{\max} = 1$ there is $U_3 \le U_2$. Moreover, instances were given with $U_3 < U_2$. On the other hand, for $\delta_{\max} > 1$ there is no strict dominance between these bounds and $U_3 > U_2$ may occur.

A reduction of the number of item types to be considered can be reached by applying the framework of Section 3.2 to (UKP). Since it is straightforward to plug in the above bounds in the same way as done for (KP) in Section 5.1.3, we refrain from going into details on this point.

The number of papers dealing with branch-and-bound approaches for (UKP) seems to be smaller than for (BKP). Early contributions by Gilmore and Gomory [175] and Cabot [59] were followed by the widely known algorithm MTU1 by Martello and Toth [323] which is a derivative of algorithm MT1 from Section 5.1.4 and a close relative of algorithm MTB for (BKP) (see Section 7.3.2). The main outline of this algorithm is given in Figure 8.6. It is assumed that the item types are sorted in decreasing order of efficiencies. All variables x_j must be initialized to 0, and the lower bound set to $z^\ell = 0$ before calling MTU1(0,0,0). Throughout the algorithm $\overline{p}$ resp. $\overline{w}$ denote the profit resp. weight of the currently packed items. The table $\underline{w}$ of minimal weights is given by $\underline{w}_j = \min_{i \ge j} w_i$. Of course, U_2 is computed in every recursive execution of MTU1 only for the corresponding subproblem.

Later a superior method was published by Martello and Toth [334]. This algorithm called MTU2 applies the *core concept* as introduced for (KP) in Section 5.4. Clearly, the basic idea of the core concept, namely that the items with high efficiency are much more likely to be included in an optimal packing than those with a low efficiency, applies even more so for (UKP) since an infinite supply of these "high-valued items" is available. After sorting the items by their efficiencies the split item

Algorithm MTU1$(j, \overline{p}, \overline{w})$:

improved := false
if $(\overline{w} > c)$ then return improved
if $(\overline{p} > z^{\ell})$ then
    $z^{\ell} := \overline{p}$, improved := true
else improved := false
if $(j > n)$ or $(c - \overline{w} < \underline{w}_j)$ or $(U_1 < z^{\ell} + 1)$ then return improved
$n_r := \lfloor (c - \overline{w}) / w_j \rfloor$
for $a := n_r$ downto 0 do
    if MTU1$(j+1, \overline{p} + ap_j, \overline{p} + aw_j)$ then
        $x_j := a$, improved := true
return improved

Fig. 8.6. Recursive Branch-and-Bound algorithm MTU1 for (UKP).

is always given by the first item for (UKP). Hence, in contrast to (KP) the core should always begin with this first item.

While the outline of MTU2 follows in principle closely the related algorithm MT2 for (KP) from Section 5.4.2, it also takes dominance into account. For the first iteration an initial core size δ_0 must be given. Experimental evidence from [334] suggests to set $\delta_0 := \max\{100, \lfloor n/100 \rfloor\}$. Note that the core C can be computed in $O(n)$ time by finding the δ-th largest (with respect to efficiencies) element in the set of item types by a modified linear median algorithm. In each of the following iterations (if any) the core size is increased by δ_0. A sketch of the main components of algorithm MTU2 is given in Figure 8.7.

Algorithm MTU2:

1. derive an upper bound U for z^*
   set $\delta := \delta_0$
   determine an approximate core $C = \{1, \ldots, \delta\}$
2. sort the items in the core
   apply dominance rules to eliminate dominated item types in C
   compute the optimal solution z_C of the core problem by MTU1
   if $z_C = U$ then
       the core solution is optimal, stop.
3. apply reduction to eliminate item types from $N \setminus C$
   if all items $j \notin C$ are eliminated then
       the core solution is optimal, stop.
4. increase the core to $C = \{1, \ldots, \delta + \delta_0\}$
   go to 2.

Fig. 8.7. Branch-and-Bound algorithm for (UKP) based on a core concept.

In the original paper by Martello and Toth [334] the upper bound U_2 from (8.21) is computed in Step 1 and multiple dominance is applied in Step 2. For every item type $j \in N \setminus C$ the reduction procedure in Step 3 works as follows. Set $x_j = 1$ and compute an upper bound for the remaining instance. If this upper bound is not larger than the current value z_C, then $x_j = 0$ holds for any solution better than the current optimal core solution (cf. [334]).

8.5 Approximation Algorithms

Following the main concepts from Sections 2.5 and 2.6 the topic of approximation for (UKP) was treated in relatively early papers usually together with approximation schemes for (KP). We will briefly review some of the corresponding results.

Algorithm U-Greedy as given in Section 8.1 is almost identical to the greedy version for (BKP). However, its worst-case performance can be bounded even without considering separately the best solution generated by a single item type. This is due to the fact that in (UKP) we can always fill at least one half of the knapsack capacity with copies of item type 1. Recall that $w_1 \leq c$. More formally we have the following theorem.

Theorem 8.5.1 *Algorithm* U-Greedy *has a relative performance guarantee of* $\frac{1}{2}$ *and this bound is tight.*

Proof. From Lemma 8.1.1 we have

$$z^* \leq z^{LP} = p_1 \frac{c}{w_1} \leq p_1 \left(\left\lfloor \frac{c}{w_1} \right\rfloor + 1 \right) \leq 2p_1 \left\lfloor \frac{c}{w_1} \right\rfloor \leq 2z^G,$$

which proves the upper bound.

To show that this bound can be reached arbitrarily close consider an instance of (UKP) with $n = 2$, $c = 2M$ and the following item types. Item type 1 is given by $w_1 = M + 1$, $p_1 = M + 2$ whereas item type 2 is defined by $w_2 = p_2 = M$. Clearly, U-Greedy achieves a total solution value of only $M + 2$ by packing a single copy of item type 1. The optimal solution consists of two copies of item type 2 with an optimal solution value of $2M$. □

A variant of U-Greedy was considered by Kohli and Krishnamurti [281]. They introduce the *total-value greedy heuristic* denoted by TV-Greedy, where the sorting of the item types follows from the hypothetical optimal solution values attained by filling in every iteration the remaining knapsack capacity with copies of only a single item type. This means that in every iteration of U-Greedy the next item type to be considered is chosen not from the sorting of the efficiencies e_j but such that the value $p_j \lfloor \overline{w}/w_j \rfloor$ is maximal. It is shown in [281] that TV-Greedy for (UKP) has a tight

relative performance guarantee of $0.59135\ldots$, which is the limit of an infinite sum of recursively defined values. Note that exactly the same algorithm was introduced and analyzed independently by White [482] who managed to bound the performance guarantee in the interval $]36/61, 42/71[$, which is $]0.59016\ldots, 0.59154\ldots[$.

The worst-case analysis of TV-Greedy indicates that its worst-case scenarios will complement those of U-Greedy. Whenever this is the case for two methods, it makes sense to combine them into a *composite heuristic* which means that instead of one algorithm, there are two algorithms performed independently. The result of a composite heuristic is given by the maximum of the two solution values.

This was done for U-Greedy and TV-Greedy by Kohli and Krishnamurti [282]. They managed to show that taking the maximum of these two greedy variants yields an approximation algorithm with a tight relative performance guarantee of $2/3$.

The construction of approximation schemes for (UKP) hardly poses any new challenges after the development of dynamic programming schemes in Section 8.3 and the construction of approximation schemes for (BKP) in Section 7.4.

The *PTAS* for (BKP) given in Section 7.4 did not take into account explicitly the bound on the number of identical copies of an item type in the construction but only as a feasibility check. The strategy of "guessing" the ℓ items with highest profit in the optimal solution by going through all sets of ℓ elements out of the ℓn possible copies of item types can be immediately applied to (UKP). Exactly the same arguments for correctness and running time are valid both for (BKP) and (UKP).

Also the *FPTAS* for (BKP) described in Section 7.4 can be adapted to (UKP) quite easily. In particular, we only have to compute the lower bound z^ℓ for (UKP) e.g. by U-Greedy and replace the dynamic programming procedure in the first part of the *FPTAS* by the dynamic programming by profits algorithm from Section 8.3.1. No other changes are necessary since the cardinality bound on the copies of item types did not play an explicit role in the *FPTAS* for (BKP).

The running time analysis yields a better result for (UKP). Since the exact dynamic programming by profits requires $O(nU)$ time, the scaled instance with $K := \varepsilon^3 z^\ell/2$ can be solved in $O(n \cdot 1/\varepsilon^3)$ time. This effort is dominated by the $O(1/\varepsilon^3)$ executions of the greedy algorithm which brings the total running time to $O(n \log n \cdot 1/\varepsilon^3)$. The space requirements are given by $O(n + 1/\varepsilon^3)$.

As before it should be noted that these complexity bounds can be easily improved upon. The remarks at the end of Section 7.4, in particular the bounds on the maximal number of large items in any feasible solution and the more careful partitioning of the profit space, would already reduce all complexities by a factor of $1/\varepsilon$. Furthermore, the numerous executions of the greedy algorithm can be replaced by a single pass through the small item types sorted in decreasing order of their efficiencies. Therefore, we have to sort the entries of the dynamic programming function by their capacities and go through them in decreasing order. Thus, the remaining

free capacity in the knapsack steadily keeps increasing and can be filled accordingly by a single pass through the small item types by the greedy algorithm. Clearly, this reduction of the second part applies also to the *FPTAS* for (BKP). With these simple improvements we immediately get a running time of $O(1/\varepsilon^2(n+\log(1/\varepsilon))+n\log n)$ with space requirements $O(n+1/\varepsilon^2)$.

In the classical paper of Lawler [295] a rather sophisticated *FPTAS* for (KP) is extended or, to be more precise, simplified to cover also (UKP). The considerable effort employed in the paper to improve the complexity of (KP) yields only an improved running time of $O(n+1/\varepsilon^3)$ for (UKP) compared to the simple scheme sketched above. The space requirements remain $O(n+1/\varepsilon^2)$.

Also the early approximation paper of Ibarra and Kim [241] contains an extension of the *FPTAS* from (KP) to (UKP) with slightly worse complexity than the scheme given in [295].

9. Multidimensional Knapsack Problems

In this first chapter of extensions and generalizations of the basic knapsack problem (KP) we will add additional constraints to the single weight constraint (1.2) thus attaining the *multidimensional knapsack problem.* After the introduction we will deal extensively with relaxations and reductions in Section 9.2. Exact algorithms to compute optimal solutions will be covered in Section 9.3 followed by results on approximation in Section 9.4. A detailed treatment of heuristic methods will be given in Section 9.5. Separate sections are devoted to two special cases, namely the two-dimensional knapsack problem (Section 9.6) and the cardinality constrained knapsack problem (Section 9.7). Finally, we will consider the combination of multiple constraints and multiple-choice selection of items from classes (see Chapter 11 for the one-dimensional case) in Section 9.8.

9.1 Introduction

It was mentioned in the Introduction in Section 1.2 that many real world applications in the selection and packing area require more than a single constraint. The resulting generalization of (KP) to the *d-dimensional knapsack problem* (d-KP) was defined as follows.

$$\text{(d-KP)} \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{9.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i = 1,\ldots,d, \tag{9.2}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n. \tag{9.3}$$

This problem can be seen as a knapsack problem with a collection of different resource constraints or one constraint consisting of a multidimensional attribute. In analogy to Section 1.4 we will impose a number of natural assumptions on the input data. As before it is assumed that all p_j, w_{ij} and c_i are positive integer values. It is allowed that $w_{ij} = 0$ for some i, j, as long as $\sum_{i=1}^{d} w_{ij} \geq 1$ holds for all items

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

$j = 1,\ldots,n$. To guarantee that an item can be packed at all we assume as a generalization of (1.14)

$$w_{ij} \leq c_i, \quad j = 1,\ldots,n,\ i = 1,\ldots,d. \tag{9.4}$$

To avoid trivial constraints it is assumed similar to (1.15)

$$\sum_{j=1}^{n} w_{ij} \geq c_i, \quad i = 1,\ldots,d. \tag{9.5}$$

Note that the terminology for (d-KP) is not completely unique in the literature. Beside the occurrence of hyphens confusion arises sometimes between the terms *multiconstraint knapsack*, *multiple knapsack* and *multidimensional knapsack*. It seems that most authors have accepted the latter name as a general term for (d-KP) and reserve *multiple* to represent the fact that an item can be put into one of *several knapsacks*. This completely different problem will be treated in Chapter 10.

We choose to use the terms *multidimensional* and *d-dimensional* in parallel where the former signifies the general model and the latter stresses the number of constraints. In this way (2-KP) follows immediately from (d-KP) and (KP) is equivalent to (1-KP).

The term *two-dimensional packing problem* also appears in a geometric context where w_{1j} and w_{2j} refer to the length and width of object j. In this case, a feasible packing has to observe the obvious geometric constraints of placing rectangles into a given rectangle of length c_1 and width c_2. These geometric problems are a major topic in the area of bin packing and will not be covered in this book. A typology for packing problems was developed by Dyckhoff [119] whereas recent surveys are given by Lodi, Martello and Monaci [303] and by Lodi, Martello and Vigo [305].

In the literature (d-KP) was not always considered as a proper member of the knapsack family of problems. For instance the book by Martello and Toth [335] does not cover (d-KP). Indeed, (d-KP) is a special case of general integer programming with the only restrictions that all coefficients are positive and the variables are zero or one. Therefore, many authors use a standard notation of linear programming involving elements of linear algebra instead of the knapsack based notation of this book.

There are two main characteristics to motivate the separate treatment of this special case. On one hand (d-KP) is a particular difficult instance of integer programming because the "constraint matrix" consisting of w_{ij} is unusually dense whereas most relevant classes of integer programming problems are defined by sparse constraint matrices. On the other hand, there is a trivial feasible solution at hand for (d-KP), namely $x_j = 0$ for all j, whereas in general integer programming finding a feasible solution can be as hard as finding an optimal solution.

Most of the earlier literature on (d-KP) in the sixties was closely connected to research on general integer programs. This may also be the reason why there still

exist no survey articles on the multidimensional knapsack problem to the best of our knowledge. A notable survey of literature was included in the paper by Chu and Beasley [83]. Moreover, one section of the survey by Lin [301] deals with (d-KP).

A plethora of research contributions investigated the general (d-KP) from the point of view of exact algorithms, heuristics, approximation and stochastic aspects. The latter will be treated in the appropriate Section 14.7. Also the special case of only two constraints, i.e. $d = 2$, was given attention to. It will be considered in Section 9.6. A further restriction of this case occurs by setting $w_{2j} := 1$ and $c_2 := k$. This yields a standard knapsack problem with an additional *cardinality constraint* meaning that at most k items can be packed into the knapsack. This topic will be treated separately in Section 9.7. The extension of the multiple-choice knapsack problem (MCKP), as introduced in Section 1.2 and fully elaborated in Chapter 11, to the multidimensional case will be briefly considered in Section 9.8.

In recent years (d-KP) turned out to be one of the favourite playgrounds for experiments with metaheuristics, in particular tabu search and genetic algorithms. It is up to discussion whether these contributions should be seen primarily as research on the multidimensional knapsack problem or whether (d-KP) only serves as a suitably "difficult" test problem for developing advanced techniques in this field. The authors are inclined to follow the second point of view. Moreover, a thorough treatment of this topic would require a detailed introduction into the basic mechanism of all these concepts including simulated annealing, threshold accepting, evolutionary algorithms, great deluge etc. which would reach far beyond the scope of this book. We will however provide some pointers to existing surveys on this area in Section 9.5.5. A further-reaching textbook on this topic was edited by Aarts and Lenstra [1].

The difficulty of (d-KP) is illustrated by the fact that the range of problem sizes where optimal solutions can be reasonably computed is still bounded roughly by $n = 500$ and $d = 10$. The theoretical foundation for this inherent difficulty will be considered in Section 9.4. From a practical point of view it should be kept in mind that most occurrences of (d-KP) involve only a relatively small number of constraints (e.g. with d in the single digits) but possibly a large number of variables.

Most of the earlier contributions to the multidimensional knapsack problem were motivated by a budget planning scenario, see e.g. the ancient work by Lorie and Savage [306] from 1955 and the papers by Weingartner [478] and Weingartner and Ness [479]. Indeed, (d-KP) is the direct mathematical formulation of a planning task of a decision maker who has to select a subset of projects (i.e. items) out of a list of possibilities, each of which (hopefully) generates a certain profit or benefit while consuming not only a monetary resource as in (KP) but a number of different types of limited resources. These can be diverse categories such as office space, available personal, vehicles, machinery, computing power or long-term and short-term financing. Along these lines Petersen [372] considered the optimal selection

of research and developement projects. More about decision problems in a financial context can be found in Sections 15.4 and 15.5.

Applications of a different flavour appear in computer science. These include the allocation of processors and databases in a distributed computer system as described by Gavish and Pirkul [165], and the scheduling of computer programs reported by Thesen [455]. In a context of business organization Yang [493] applied a (d-KP) model to the problem of allocating shelf space to consumer products in a retail store. Other fields of planning and optimization, where (d-KP) frequently appears in the mathematical formulation, are the loading of cargo (cf. Shih [434]), cutting stock problems and the scheduling of tasks or projects.

9.2 Relaxations and Reductions

Basically, all statements of this section apply for general integer programming problems but are presented for the case of the multidimensional knapsack problem. Whereas for (KP) efficient upper bounds could be derived by adding heuristic arguments to the solution of the LP-relaxation (see Section 5.1.1), different relaxations were applied for (d-KP) on which upper bounds could be based. As for all integer programming problems we will start by considering the LP-relaxation of (d-KP) given by replacing $x_j \in \{0,1\}$ with $x_j \in \mathbb{R}$ for all $j = 1,\ldots,n$ and yielding the optimal solution value z^{LP}. The structure of an optimal solution vector x^{LP} can be characterized by the following straightforward observation which was mentioned by many authors in the past.

Lemma 9.2.1 *There exists an optimal solution vector x^{LP} with at most $\min\{d,n\}$ fractional values.*

Proof. Assume $d < n$ (otherwise the statement is trivial). The LP-relaxation of (d-KP) is a linear program with $n+d$ constraints and $2n+d$ variables (including slack variables). Any basic feasible solution of this linear program contains at least n zero values. Of these n values at most d may belong to slack variables corresponding to a capacity constraint (9.2). For every one of the remaining at least $n-d$ zero values there are two possibilities. Either one of the original variables $x_1,\ldots,x_n$ is 0, in which case it is clearly not fractional, or one of the slack variables corresponding to a constraint $x_j \leq 1$ is 0. But then we have $x_j = 1$ which is not fractional again. Hence, at least $n-d$ original variables turn out to be 0 or 1 and at most d variables remain to have possibly fractional values. □

Clearly this statement holds for every linear program with n variables from $[0,1]$ and d constraints. Contrary to (KP) there is no easy way to compute the solution of the LP-relaxation. It was shown by Megiddo and Tamir [343] that x^{LP} can be computed in $O(n)$ time if d is assumed to be constant (recall from the introduction that d is rather small in most applications of (d-KP)).

The techniques of *Lagrangian relaxation* and *surrogate relaxation* were introduced in Section 3.8 where (d-KP) served as an example to illustrate these methods. Since the bounds derived from these relaxations play an important role for the optimal solution methods of (d-KP) and also for some of the heuristic approaches, we will go into more detail. For notational convenience let $\mathbb{R}^m_+ := \{y \in \mathbb{R}^m \mid y_j \geq 0 \text{ for } j = 1,\ldots,m\}$ for any $m \in \mathbb{N}$.

Let us recall some of the definitions given in Section 3.8. If the context is clear we omit the letter P denoting the problem to be relaxed and briefly write $z(L(\lambda,M))$ instead of $z(L(P,\lambda,M))$. For a subset of constraints $M \subseteq \{1,\ldots,d\}$ with $m := |M|$ the Lagrangian relaxation of (d-KP) with optimal solution value $z(L(\lambda,M))$ for a set of Lagrangian multipliers $\lambda \in \mathbb{R}^m_+$ is given as follows.

$$z(L(\lambda,M)) = \text{maximize} \sum_{j=1}^{n} \left(p_j - \sum_{i \in M} \lambda_i w_{ij} \right) x_j + \sum_{i \in M} \lambda_i c_i \tag{9.6}$$

$$\text{subject to} \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i \in \{1,\ldots,d\} \setminus M, \tag{9.7}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

The computation of $z(L(\lambda,M))$ for a fixed multiplier λ is again an instance of (d-KP) but with a reduced number of constraints. Typically, m is chosen close to d such that $z(L(\lambda,M))$ can be found by solving an instance of (KP) or (2-KP). It was pointed out before that every feasible solution of (d-KP) is feasible for the Lagrangian relaxation and increases the objective function value thus generating an upper bound $z(L(\lambda,M)) \geq z^*$ for every M. The classical reference for the more general application of Lagrangian relaxation is Fisher [147], whereas its origin in this area can be attributed to Everett [135].

The *surrogate relaxation* simply aggregates constraints by replacing them by their weighted sum. The use of this relaxation goes back to the seminal works by Glover [178] and [179] who established a trove of theoretical results. As introduced in Section 3.8, a set of constraints is selected with the same notation as above and surrogate multipliers $\mu \in \mathbb{R}^m_+$ are applied.

$$z(S(\mu,M)) = \text{maximize} \sum_{j=1}^{n} p_j x_j \tag{9.8}$$

$$\text{subject to} \sum_{i \in M} \mu_i \left(\sum_{j=1}^{n} w_{ij} x_j \right) \leq \sum_{i \in M} \mu_i c_i, \tag{9.9}$$

$$\sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i \in \{1,\ldots,d\} \setminus M, \tag{9.10}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

Naturally, all feasible solutions for (d-KP) are also feasible for the surrogate relaxation and therefore $z(S(\mu,M)) \geq z^*$ for every M and μ. Since the objective functions (9.1) and (9.8) are identical it follows immediately that if the optimal solution of the surrogate relaxation for some M and μ is feasible for (d-KP), then this solution is optimal for (d-KP). This observation, which does not hold for the Lagrangian relaxation was described by Glover [181] as "duality theorem" for surrogate mathematical programming.

Again, the computation of the solution value $z(S(\mu,M))$ for a fixed multiplier μ is an instance of (d-KP). However, in the case of surrogate relaxation one frequently sets $M = \{1,\ldots,d\}$ and thus only an instance of (KP) has to be solved.

A combination of these two relaxations is given by the *composite relaxation* denoted by $Co(\lambda,\mu)$ which goes back to Greenberg and Pierskalla [198]. By doing both relaxations at the same time one hopes to relax the constraints thus making the computation easier while at the same time using the penalty concept of Lagrangian relaxation to keep the solutions "near" the feasible domain. For simplicity of notation we follow the original approach from [198] and consider only the case $M = \{1,\ldots,d\}$.

$$z(Co(\lambda,\mu)) = \text{maximize} \sum_{j=1}^{n} \left(p_j - \sum_{i=1}^{d} \lambda_i w_{ij} \right) x_j + \sum_{i=1}^{d} \lambda_i c_i \tag{9.11}$$

$$\text{subject to} \sum_{i=1}^{d} \mu_i \left(\sum_{j=1}^{n} w_{ij} x_j \right) \leq \sum_{i=1}^{d} \mu_i c_i, \tag{9.12}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

Obviously, the optimal solution value $z(Co(\lambda,\mu))$ for fixed λ and μ is determined by solving an instance of (KP). The Lagrangian and surrogate relaxations are special cases of the composite relaxation.

Clearly, the more complicated composite relaxation will yield better (i.e. lower) solution values than the LP-relaxation. However, the amount of this improvement is not unlimited but subject to upper bounds which were given by Crama and Mazzola [93].

Theorem 9.2.2 *For all $\lambda,\mu \in \mathbb{R}^m_+$ there is*

$$z^{LP} - p_{\max} \leq z(Co(\lambda,\mu)), \tag{9.13}$$

$$\frac{1}{2} z^{LP} \leq z(Co(\lambda,\mu)). \tag{9.14}$$

Proof. Following the proof in [93] let the largest coefficient in the objective function (9.11) for every λ be given by $p(\lambda) := \max\{p_j - \sum_{i=1}^{d} \lambda_i w_{ij} \mid j = 1,\ldots,n\}$ attained for item $j(\lambda)$. Consider the LP-relaxation of the composite relaxation (9.11),

(9.12), $x_j \in [0,1]$ for $j = 1,\ldots,n$, with an optimal solution value of $z(C(Co(\lambda,\mu)))$. By comparing the set of feasible solutions it follows immediately that

$$z^{LP} \leq z(C(Co(\lambda,\mu))). \tag{9.15}$$

Now we can distinguish between two cases. If $p(\lambda) > 0$ then the knapsack structure of the composite relaxation can be exploited. Applying Corollary 2.2.2 (and neglecting all items with negative coefficients) we get that

$$z(C(Co(\lambda,\mu))) \leq z(Co(\lambda,\mu)) + p(\lambda)$$

and from Lemma 2.2.4

$$\frac{1}{2} z(C(Co(\lambda,\mu))) \leq z(Co(\lambda,\mu)).$$

Together with (9.15) and the fact that $p(\lambda) \leq p_{j(\lambda)} \leq p_{\max}$ the claim follows.

The second case of $p(\lambda) \leq 0$ means that all coefficients in (9.11) are nonpositive. Hence, the optimal solution will trivially consist of not packing any items and

$$z(Co(\lambda,\mu)) = z(C(Co(\lambda,\mu))) = \sum_{i=1}^{d} \lambda_i c_i .$$

With (9.15) we get immediately

$$z^{LP} \leq z(Co(\lambda,\mu))$$

which is even stronger than the claim of the theorem. □

For the computation of upper bounds on z^* one would clearly wish to have multipliers for the different types of relaxations which yield the smallest possible upper bound. The corresponding optimization problem was introduced in (3.20) of Section 3.8 as the *Lagrangian dual problem*. For the sake of completeness we will define all three dual problems resulting from the above relaxations.

$$\begin{aligned} z(LD(M)) &= \min\{z(L(\lambda,M)) \mid \lambda \geq 0\} \\ z(SD(M)) &= \min\{z(S(\mu,M)) \mid \mu \geq 0\} \\ z(CoD) &= \min\{z(Co((\lambda,\mu) \mid \lambda \geq 0, \mu \geq 0\} \end{aligned} \tag{9.16}$$

An important characterization of the Lagrangian relaxation is given by the following theorem from Nemhauser and Wolsey [360, ch.II.3.6], which was also mentioned at the end of Section 3.8.

Theorem 9.2.3 *For every $M \subseteq \{1,\ldots,d\}$ there is*

$$z(LD(M)) \leq z^{LP}.$$

Proof. (Sketch) Let $Q(M) := \{x \in \{0,1\}^n \mid \sum_{j=1}^{n} w_{ij}x_j \leq c_i \text{ for } i \in \{1,\ldots,d\} \setminus M\}$. By duality theory $z(LD(M))$ can also be attained with

$$\begin{aligned} z(LD(M)) = \text{ maximize } & \sum_{j=1}^{n} p_j x_j \\ \text{subject to } & \sum_{j=1}^{n} w_{ij}x_j \leq c_i, \quad i \in M, \\ & x \in \text{ conv}(Q(M)), \end{aligned}$$

where $\text{conv}(Q(M))$ denotes the convex hull of $Q(M)$. Since every feasible solution of this linear program is also feasible for the LP-relaxation the result follows. □

A comparison of the four values resulting from the discussed relaxations is given in the following theorem which is mostly a combination of several well-known facts (see e.g. [166]).

Theorem 9.2.4 *For $M = \{1,\ldots,d\}$ the following inequalities hold:*

$$z^* \leq z(CoD) \leq z(SD(M)) \leq z(LD(M)) \leq z^{LP}$$

Proof. We will go through the four inequalities from left to right. The first inequality is trivial since every feasible solution for (d-KP) is also feasible for the composite relaxation independently from the multipliers. Moreover, the objective function value of a feasible solution in (9.11) will never be smaller than in (d-KP).

The second inequality is straightforward since the surrogate relaxation is the special case of the composite relaxation with $\lambda_1 = \ldots = \lambda_d = 0$.

To show the third inequality we can observe that for every fixed set of multipliers μ the objective function (9.6) for $\lambda = \mu$ will be at least as large as (9.8) for the same feasible solution. Since every feasible solution of the surrogate relaxation with $M = \{1,\ldots,d\}$ is also feasible for the Lagrangian relaxation this means that $z(S(\mu,M)) \leq z(L(\mu,M))$ for every $\mu \in \mathbb{R}_+^d$ which proves the claim.

The final inequality is a special case of Theorem 9.2.3. □

A direct conclusion can be drawn from Theorems 9.2.2 and 9.2.4 (cf. [93]).

Corollary 9.2.5 *For $M = \{1,\ldots,d\}$ there is*

$$\begin{aligned} z^{LP} - p_{\max} \leq z(CoD) \leq z(SD(M)) \leq z(LD(M)) \leq z^{LP}, \\ \frac{1}{2} z^{LP} \leq z(CoD) \leq z(SD(M)) \leq z(LD(M)) \leq z^{LP}. \end{aligned}$$

Computational experiments about the practical behaviour of different bounds resulting from various relaxations were performed by Gavish and Pirkul [166]. Their empirical evidence is consistent with Theorem 9.2.4 and suggests that bounds from surrogate relaxations are clearly better than from Lagrangian relaxation whereas the further improvement brought about by the composite relaxation is only modest.

The computation of the dual solution values (9.16), i.e. of the optimal multipliers, turns out to be rather challenging in practice. A great deal of research has been done on this problem, mostly considering general integer programming problems. It is beyond the scope of this volume to give a comprehensive treatment of this subject. A detailed study of the computation of multipliers concentrating on the multidimensional knapsack problem was given by Gavish and Pirkul [166]. Their work also includes a wealth of related references.

For the Lagrangian dual problem the widely accepted opinion suggests the use of *subgradient* optimization which seems to yield stable bounds of reasonable quality however with higher computational effort. The basic framework of this method consists of an iteration rule for λ^k, $k = 1,\ldots,$ with a step width t_k:

$$\lambda_i^{k+1} := \lambda_i^k + t_k \left(\sum_{j=1}^{n} w_{ij} x_j(\lambda^k) - c_i \right), \quad \text{for } i = 1,\ldots,d, \tag{9.17}$$

where $x(\lambda^k)$ is the optimal solution of the Lagrangian relaxation for λ^k. Given a feasible solution with value z^ℓ, Gavish and Pirkul [166] use a step width of

$$t_k := \delta_k \cdot \frac{z(L(\lambda^k, M)) - z^\ell}{\sqrt{\sum_{i=1}^{d} \left(\sum_{j=1}^{n} w_{ij} x_j(\lambda^k) - c_i \right)^2}}. \tag{9.18}$$

The constant $\delta_k \in [0,2]$ is determined by starting with $\delta_0 = 2$ and reducing the value to $\delta_{k+1} := \delta_k/2$ if the bound $z(L(\lambda^k, M))$ did not improve during the last 15 iterations. In their experiments they set $|M| = d - 1$ and select the single remaining constraint such that it produces the lowest bound if the problem is considered as an instance of (KP) with only this particular constraint.

A quasi-subgradient method to compute surrogate multipliers was presented in detail by Fréville and Plateau [154]. Other search procedures for the surrogate dual problem were given e.g. by Karwan and Radin [259], Dyer [121] and Gavish and Pirkul [166]. We will briefly outline the latter. The authors first present a procedure to compute an approximation of the optimal surrogate multipliers for problem (2-KP) (see also Section 9.6). W.l.o.g. the multiplier μ_1 for the first constraint is set to 1 and an upper and lower bound for the optimal multiplier μ_2 are chosen. Then an iterative bisection procedure reduces the gap between these bounds and stops if an ε–neighborhood of the optimal second multiplier is attained.

To determine multipliers for (d-KP) this procedure is used as a subroutine during a number of iterations. The two constraints to be used in every iteration are chosen as follows. The first constraint is the one which produces the lowest bound if all other constraints are neglected (see above). Inserting the solution of this (KP) instance into the original (d-KP) the most violated constraint is chosen as second constraint. In the next iteration only the second constraint remains and is complemented again by the constraint most violated by the corresponding (KP) solution. Through this selection of two "contradicting" constraints in every iteration one can hope to reach a reasonable representation of the feasible domain.

Instead of solving a knapsack problem to optimality in every iteration one frequently settles for an approximate solution.

To find "efficient" multipliers for the composite relaxation it was suggested by Gavish and Pirkul [166] to compute λ and μ separately starting with the latter. Therefore it is recommended to initialize $\lambda_1 = \ldots = \lambda_d = 0$ and determine surrogate multipliers as described above. When these are fixed, Lagrangian multipliers are computed quite easily for the remaining problem, e.g. by a subgradient optimization procedure.

Further results about the dual problems of the relaxations were given e.g. by Greenberg and Pierskalla [198], Glover [181] and Karwan and Rardin [259]. Recall that almost all literature in this area deals with general integer programming problems.

The concept of *dominance*, which proved to be very important for (UKP) (see Section 8.2.2) but is also used as a preprocessing tool for (KP) (see Section 3.4), is less successful for reducing the size of a (d-KP) but computationally much more expensive. Indeed, item j dominating item k would require that $p_j \geq p_k$ and $w_{ij} \leq w_{ik}$ for all $i = 1,\ldots,d$.

The strategy of *reduction* of a given problem instance was introduced for (KP) in Section 3.2. Also in general integer (and especially binary) programming it is a classical preprocessing step to try to fix the values of some of the variables before the actual solution procedure is started, or at least to impose some bounds on them.

The general idea of this approach can be described as follows. For a variable x_k, $k \in \{1,\ldots,n\}$, a subset of constraints $M \subseteq \{1,\ldots,d\}$ and a binary value $b \in \{0,1\}$ consider the following subproblem of (d-KP).

$$z_k = \text{maximize} \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i \in M, \tag{9.19}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n,\ j \neq k, \tag{9.20}$$

$$x_k = b.$$

Assume some feasible solution $\bar{x}$ is known with an objective function value $\bar{z} \leq z^*$. A straightforward argument analogous to Section 3.2 based on the fact that no solution with $x_k = b$ can be better than z_k shows the following fact:

> If $z_k \leq \bar{z}$ then either $\bar{x}$ is an optimal solution of (d-KP)
> or $x_k = 1 - b$ in every optimal solution.

Unfortunately, the above integer program is again an instance of (d-KP) with $n-1$ variables and $|M|$ constraints. Thus a trade-off appears between taking into account a large set M yielding better reduction results at a higher computational cost versus considering only single constraints at a time ($|M| = 1$) which is much faster to do (since it amounts to the solution of a (KP) instance) but yields inferior reductions.

Note that the optimal solution value z_k could be replaced by any upper bound on z_k. Different strategies were developed in the literature to find such upper bounds which preserve the reduction of items as far as possible. In particular, the papers by Fayard and Plateau [137] and Fréville and Plateau [154] (with its predecessor [152]) should be mentioned. The latter contains a rather sophisticated framework called RAMBO to fix the value of variables and also to eliminate constraints. The full details of the iterative process which exploits Lagrangian relaxation in an efficient way are found in [154].

A compromise between a computationally expensive reduction with a large number of constraints and a less effective one with a single constraint is attained by the *pairing of constraints* as introduced by Hammer, Padberg and Peled [208]. In their approach linear combinations of pairs of constraints resembling surrogate relaxations are used to put tighter bounds on the values of integer variables, respectively to fix the values of binary variables.

A special version of constraint pairing was elaborated in Osorio, Glover and Hammer [363]. In their version of a surrogate relaxation the n inequalities $x_j \leq 1$ are added to the d capacity constraints. Then the LP-relaxation of the dual of the resulting problem is computed yielding an optimal solution $\mu \in \mathbb{R}_+^{d+n}$. This vector is used as a surrogate multiplier to aggregate the $d+n$ conditions into a single constraint

$$\sum_{i=1}^{d} \mu_i \left(\sum_{j=1}^{n} w_{ij} x_j \right) + \sum_{j=1}^{n} \mu_{d+j} x_j \leq \sum_{i=1}^{d} \mu_i c_i + \sum_{j=1}^{n} \mu_{d+j}. \tag{9.21}$$

The right-hand side $U := \sum_{i=1}^{d} \mu_i c_i + \sum_{j=1}^{n} \mu_{d+j}$ is precisely the objective function value of the LP-relaxation of the dual problem computed before. By strong duality we get $U = z^{LP} \geq z^*$. A similar approach was performed by Gavish and Pirkul [166].

Furthermore, any feasible integer solution can be used to derive a lower bound z^ℓ on z^* which can be written as a trivial constraint

$$\sum_{j=1}^{n} p_j x_j \geq z^{\ell}. \tag{9.22}$$

For convenience denote the coefficient of x_j in (9.21) by $s_j := \sum_{i=1}^{d} \mu_i w_{ij} + \mu_{d+j}$. Now the two constraints (9.21) and (9.22) can be *paired* and written as

$$\sum_{j=1}^{n} (s_j - p_j) x_j \leq U - z^{\ell}. \tag{9.23}$$

Since μ was chosen as an optimal solution vector of the LP-relaxation of the dual problem we have from complementary slackness that for every index j with $x_j^{LP} > 0$ in the optimal solution of the primal LP-relaxation, there must be $s_j = p_j$. Hence, we can eliminate all such variables from (9.23) and get a stronger inequality.

A straightforward utilization of this strengthened constraint for reduction is given by setting $x_j := 0$ if $s_j - p_j > U - z^{\ell}$. Therefore, one would obviously be interested in decreasing the right-hand side of (9.23). For U there is no such possibility since it was derived from the original LP-relaxation. However, any available lower bound on z^* can be plugged in as z^{ℓ} to improve the chances of fixing variables to 0. In [363] this construction and the resulting inequality is also used to derive logic cuts for integer programming problems.

9.3 Exact Algorithms

The main approach for computing an optimal solution of (d-KP) is the branch-and-bound method starting with very early contributions in the sixties. However, some attempts were also made to tackle the problem by dynamic programming. These will be considered in Section 9.3.2.

9.3.1 Branch-and-Bound Algorithms

It should be mentioned once more at this point that classical branch-and-bound algorithms for general integer programming were frequently applied also to (d-KP). Since it is beyond the scope of this book to cover this area completely we will restrict ourselves to some papers dealing mainly or exclusively with (d-KP).

A special algorithm dedicated to (d-KP) with unbounded integer variables was given by Cabot [59]. It invokes Fourier-Motzkin elimination to derive bounds on the value of single variables.

The earliest paper containing a branch-and-bound algorithm especially for (d-KP) and following a notion of branch-and-bound as it is seen as standard model today

(cf. Section 2.4) was given by Thesen [456] in 1975. It follows a straightforward approach and concentrates in particular on saving core memory space.

A more elaborate approach was given by Shih [434]. In this paper the necessary upper bounds were derived by considering the following family of d standard knapsack problems containing only the i-th constraint for some $i \in \{1,\dots,d\}$.

$$(KP)_i \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{9.24}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_{ij} x_j \leq c_i,$$

$$x_j \in \{0,1\}, \quad j = 1,\dots,n.$$

Let z_i^* by the optimal solution value of $(KP)_i$. Then a straightforward upper bound on the optimal solution value of (d-KP) is given by

$$z^* \leq U_1 := \min\{z_i^* \mid i = 1,\dots,d\},$$

because every z_i^* is an upper bound for z^*. Since the computation of an optimal solution for d different instances of (KP) is still unacceptable in a branch-and-bound framework, z_i^* was replaced by the optimal solution value of the LP-relaxation of $(KP)_i$ denoted by z_i^{LP}. This yields a weaker but efficiently computable upper bound

$$z^* \leq U_1 \leq U_2 := \min\{z_i^{LP} \mid i = 1,\dots,d\}.$$

The most efficient currently published branch-and-bound algorithm for (d-KP) seems to be due to Gavish and Pirkul [166]. Their paper is highly recommended to anybody interested in optimal solution methods for (d-KP). Before starting the branch-and-bound enumeration they perform reductions following to some extent the ideas described in Section 9.2. Detailed studies on the effectiveness of reductions for different types of (d-KP) instances are included in [166].

Following the general branch-and-bound model from Section 2.4 an extensive computational study of the different modules in such an algorithm was performed. Since the computation of an upper bound is much more time consuming for (d-KP) than for (KP) the authors also include the possibility of *not* computing a new bound in every node of the branch-and-bound tree. The upper bounds were computed by the surrogate relaxation as introduced in Section 9.2. Different strategies for updating the set of surrogate multipliers throughout the execution of the algorithm were compared. Also the separation of the solution space, i.e. the selection of the branching variable, and the order of processing the nodes of the branch-and-bound tree were investigated in [166]. Moreover, during the execution of the algorithm additional reduction steps can be performed to fix the values of variables. An extension of the applied methods from the binary to the bounded case can be found in Pirkul and Narasimhan [379].

An iterative approach which solves a sequence of linear programs derived from the LP-relaxation of (d-KP) was presented by Soyster, Lev and Slivka [443]. The construction of the successive linear programs is related to the classical cut algorithm of Gomory for integer programming. Their algorithm is tuned especially for instances where n is large but d is quite small. In these cases the number of fractional values in a basic optimal solution of the LP-relaxation of (d-KP) is also small since Lemma 9.2.1 bounds this quantity by d. Computational experiments indicate that the proposed algorithm is not extremely successful in computing optimal solutions for every instance but can be expected to find feasible solutions very close to the optimal value within reasonable time.

9.3.2 Dynamic Programming

The early attempts to tackle (d-KP) by dynamic programming resulted in a number of papers such as Gilmore and Gomory [177], Weingartner and Ness [479], Nemhauser and Ullmann [358]. In these historical contributions, (d-KP) was generally derived from a budgeting background.

The dynamic programming function can be stated as a straightforward generalization from (KP) as introduced in Section 2.3. For $j = 0,\dots,n$ and $g_i = 0,\dots,c_i$ for $i = 1,\dots,d$ let $z(j,g_1,\dots,g_d)$ be the optimal solution value of the following subproblem of (d-KP) consisting of item set $\{1,\dots,j\}$ and reduced knapsack capacities $g_1,\dots,g_d$ analogous to (2.7)

$$\begin{aligned} \text{maximize} \quad & \sum_{\ell=1}^{j} p_\ell x_\ell \\ \text{subject to} \quad & \sum_{\ell=1}^{j} w_{i\ell} x_\ell \le g_i, \quad i = 1,\dots,d, \\ & x_\ell \in \{0,1\}, \quad \ell = 1,\dots,j. \end{aligned} \tag{9.25}$$

The trivial extension of (2.8) to compute the values of $z(j,g_1,\dots,g_d)$ is given by the following recursion.

$$z(j,g_1,\dots,g_d) := \begin{cases} z(j-1,g_1,\dots,g_d) \\ \qquad \text{if } g_i < w_{ij} \text{ for some } i \in \{1,\dots,d\}, \\ \max\{z(j-1,g_1,\dots,g_d),\ p_j + z(j-1,g_1-w_{1j},\dots,g_d-w_{dj})\} \\ \qquad \text{if } g_i \ge w_{ij} \text{ for all } i \in \{1,\dots,d\}. \end{cases} \tag{9.26}$$

Computing the optimal solution value $z(n,c_1,\dots,c_d)$ can be seen as filling a table of size $n \cdot c_1 \cdot \ldots \cdot c_d$, i.e. $O(nc_{\max}^d)$, which is clearly a huge task with respect to time and space. Hence, it is no surprise that the computational success of dynamic

programming algorithms from the early papers mentioned above was quite limited, although the number of constraints in most computational tests was only $d = 2$.

In our personal taste the most interesting paper from this area is the one by Weingartner and Ness [479] which offers also a "dual" approach. In this complement problem all items are included at the beginning. During the dynamic programming iterations items are removed resp. unpacked until a feasible solution is found. Note that similar strategies were pursued for greedy algorithms (see Section 9.5.1). In [479] also reductions through lower and upper bounds and implementational details are presented.

For the unbounded version of (d-KP) there is an advanced algorithm by Ozden [364] for problems with a small number of constraints ($d \leq 5$) based on the concept of DP-with-Lists. It applies a straightforward generalization of the simple dominance from Section 8.2.2 as a dominance test in a preprocessing phase to reduce the number of variables. Since the algorithm is explicitly restricted to small d, the computational effort of this dominance test remains acceptable. For (d-KP) the number of states computed during the dynamic programming iterations grows rapidly even for moderate d. Therefore, special attention is paid in [364] to apply refined versions of state reduction through dominance. Furthermore, along the lines of Section 3.5 upper bounds are computed by the surrogate dual problem to remove unpromising states.

Naturally, the computational effort for the upper bound computation increases throughout the iterations. On the other hand, we would like to have even tighter bounds and thus more effective reductions when the number of states increases. This deadlock situation is broken by Ozden [364] through a partitioning of the given problem instance. In principle, the set of items N can be partitioned into mutually disjoint subsets. Then the list of dynamic programming states (including the zero state) is computed for each of them. Finally, the optimal solution must be a combination including exactly one state for each subset. The optimal combination of states is in fact an instance of the multidimensional multiple-choice knapsack problem which will be discussed in Section 9.8. This decomposition approach is related to the storage reduction scheme of Section 3.3 although a bottom up strategy is applied here whereas a top down approach was performed in the latter. It should be noted that in [364] the different subproblems are not handled completely independent since information of previously considered subproblems, in particular lower bounds, can be exploited.

The implementation of [364] limits the number of subproblems, i.e. different classes in the multiple-choice problem, to at most 5. The partitioning into subproblems is invoked dynamically whenever the number of states exceeds a certain threshold. Computational experiments show that the smaller the number of constraints, the more effective the preprocessing step turns out to be. Problem (UKP) could be seen as a limit point of this process. Altogether, the reported results exhibit a convincing behaviour of this method for small values of d.

Recently, two new attempts were made to develop competitive algorithms based on dynamic programming. Both of them use upper bounds (cf. Section 3.5) and fixing of variables to reduce the size of the considered subinstances. The first one is due to Bertsimas and Demir [34]. It applies a sort of backward recursion, where x_n is determined according to (9.26) but not by comparing the unknown exact values of $z(n-1,c_1,\ldots,c_d)$ and $p_n+z(n-1,c_1-w_{1n},\ldots,c_d-w_{dn})$, but by exploiting upper and lower bounds on these solution values of subinstances. Therefore, the resulting solution of this *approximate dynamic programming* approach is not optimal but only an approximation and will be discussed in Section 9.5.4. An interesting feature in this paper is the construction of estimations for an entry $z(j,g_1,\ldots,g_d)$ by, loosely speaking, "weighted interpolation" from the entries corresponding to a set of sample capacity vectors.

A different approach to improve upon the straightforward dynamic programming algorithm was given by Balev, Yanev, Fréville and Andonov [24]. Instead of storing the huge dynamic programming table a list representation of the dynamic programming states is used analogous to Section 3.4. This means that only those entries of the dynamic programming function z are stored where the function value changes. Unlike for (KP), dominance between the generated states occurs only seldom and is hardly worth the computational effort.

A more successful approach to speed up the cumbersome performance of dynamic programming is the utilization of upper bounds. Using upper bounds to improve dynamic programming algorithms was done before for (KP) e.g. in the Minknap algorithm by Pisinger [383] described in Section 5.4. The method suggested in [24] is different. It computes a sequence of upper and lower bounds in the following way.

For a given feasible solution vector $\bar{x}=(\bar{x}_1,\ldots,\bar{x}_n)$ we consider n subproblems. In the kth subproblem, $k=1,\ldots,n$, the value of variable x_k is fixed to the opposite value as in the feasible solution $\bar{x}$. This means that we negate the decision made on the packing of item k and get an instance of (d-KP) where (9.3) is replaced by

$$x_j \in \{0,1\},\ j=1,\ldots,n,\ j\neq k, \quad x_k = 1-\bar{x}_k. \tag{9.27}$$

For the resulting problem an upper bound u_k is determined. Instead of computing the values of the dynamic programming function in the natural order of items from 1 to n, the items resp. variables are considered in decreasing order of the n upper bounds $u_1,\ldots,u_n$. In this way, one hopes to deal with those variables first which promise to have a larger influence on the optimal solution value and thus have a smaller number of changes in the dynamic programming function later on. For notational convenience we will assume for the reminder of this description that the items are renumbered such that $u_1 \geq u_2 \geq \ldots \geq u_n$.

Beside these upper bounds we also compute lower bounds during the dynamic programming computation. In particular, after considering an item k and computing $z(k,c_1,\ldots,c_d)$ we calculate a lower bound ℓ_k by setting the remaining variables

equal to the known feasible solution, i.e. $x_{k+1} = \bar{x}_{k+1}, \ldots, x_n = \bar{x}_n$ and filling the remaining capacities in the best possible way. This partial optimal solution can be found immediately from the dynamic programming table.

It is easy to see that the resulting sequence of lower bounds is increasing, i.e. $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n = z^*$, since we gain one degree of freedom in every iteration. The following Lemma from [24] shows that we have reached optimality as soon as the lower bounds "overtake" the upper bounds.

Lemma 9.3.1 *If $\ell_k \geq u_{k+1}$, then $z^* = \ell_k$.*

Proof. If the part of the feasible solution $\bar{x}_{k+1}, \ldots, \bar{x}_n$ used for computing ℓ_k is identical to the corresponding part of an optimal solution, then the statement follows immediately from the definition of ℓ_k which guarantees an optimal completion of this partial solution.

If $x_h^* = 1 - \bar{x}_h$ for some $h = k+1, \ldots, n$, then there is

$$z^* \leq u_h \leq u_{k+1} \leq \ell_k \leq z^*,$$

which means that all inequalities must be fulfilled with equality. □

Plugging in the computation of the upper and lower bounds and exploiting Lemma 9.3.1 can be done in a straightforward way by adapting DP-with-Lists from Section 3.4 (see [24] for more details). Since it is unlikely that the stopping criterion induced by Lemma 9.3.1 will be valid for very small values of k, the evaluation of the lower bounds can be omitted in the earlier iterations.

Obviously, finding a feasible solution $\bar{x}$ and computing the upper bounds $u_1, \ldots, u_n$ is crucial for the performance of the algorithm. In [24] the bounds were derived by rounding down the solution value of the linear programming relaxation but other relaxations (see Section 9.2) could be plugged into the algorithm just as well. Also for determining $\bar{x}$ the solution vector x^{LP} of the LP-relaxation of (d-KP) is used. Note that if $\bar{x}_j = 1 - x_j^{LP}$ for some $j = 1, \ldots, n$, then the fixation $x_j = 1 - \bar{x}_j = x_j^{LP}$ has no effect on the computation of the upper bound u_j. Hence, it is reasonable to set $\bar{x}_j = x_j^{LP}$ whenever x_j^{LP} is non-fractional. To determine those entries of $\bar{x}$ where x^{LP} is fractional, the authors of [24] suggest to solve a reduced instance of (d-KP) where the other entries are fixed as prescribed above. Since there can be at most d fractional variables in x^{LP} (see Lemma 9.2.1) and d is usually much smaller than n, the resulting instance of (d-KP) is quite small and can be expected to be solved to optimality (e.g. be recursive application of the given algorithm). Other heuristics to compute feasible solutions are discussed in Section 9.5.

The resulting algorithmic framework is reported to be very fast in the earlier phases of the computation but then slows down considerably. However, it offers a very efficient preprocessing tool to fix the value of many variables and thus leaves an instance of reduced size to be solved. Because the upper bounds will be strengthened by the fixing of variables, it makes sense to apply the algorithm again to the reduced

instance and solve (d-KP) to optimality by a recursion. The corresponding preliminary computational results by Balev et al. [24] indicate a promising behaviour of the overall algorithm.

9.4 Approximation

It was pointed out in the previous section that finding an optimal solution of (d-KP) is computationally very expensive. Hence, the development of efficient approximation schemes would be particularly welcome. Unfortunately, theoretical results rule out the existence of fully polynomial time approximation schemes (*FPTAS*) under the usual assumption as shown in the following section. Therefore, all we can hope for is the development of polynomial time approximation schemes (*PTAS*) which will be elaborated in Section 9.4.2. Other heuristic algorithms which are more successful from a practical point of view but usually lack a worst-case performance guarantee will be discussed in Section 9.5.

9.4.1 Negative Approximation Results

It was shown by Gens and Levner [168] in 1979 and independently by Korte and Schrader [286] in 1981 that the existence of a fully polynomial time approximation scheme even for (2-KP) would imply $\mathcal{P}=\mathcal{NP}$, i.e. that every $\mathcal{NP}$-hard optimization problem could be solved by a polynomial time algorithm. Since hardly anybody believes this identity to be true, there is no hope for the construction of an *FPTAS* for (d-KP).

Theorem 9.4.1 *There is no* FPTAS *for (2-KP) unless* $\mathcal{P}=\mathcal{NP}$.

Proof. Since the original references [168] and [286] are not easy to access, we include a self-contained and simplified proof (based on the construction in [286]) assuming that the interested reader is familiar with the fundamental mechanism of $\mathcal{NP}$-completeness proofs (see also Appendix A).

Let us first introduce the classical *equipartition problem.*

EQUIPARTITION
Instance: n items with integer weight w_j for $j = 1,\ldots,n$ and n even.
Question: Is there a subset of items S with

$$|S| = n/2 \quad \text{and} \quad \sum_{j\in S} w_j = \sum_{j\notin S} w_j \; ? \tag{9.28}$$

It is well known that EQUIPARTITION is $\mathcal{NP}$-complete (see [164, SP12]). To show the $\mathcal{NP}$-completeness of (d-KP) we will define the following decision version of (2-KP) with a cardinality objective function.

CARDINALITY (2-KP)
Instance: n items each with 2 integer weights w_{1j}, w_{2j} for $j = 1,\ldots,n$, capacity bounds c_1, c_2, a cardinality bound K.
Question: Is there a subset of items S with

$$|S| \geq K, \tag{9.29}$$

$$\sum_{j\in S} w_{1j} \leq c_1, \tag{9.30}$$

$$\sum_{j\in S} w_{2j} \leq c_2 \ ? \tag{9.31}$$

Given any instance of EQUIPARTITION we will construct a corresponding instance of CARDINALITY (2-KP) such that both questions always yield the same answer for a pair of instances.

For an instance I of EQUIPARTITION define as before $w_{\max} := \max\{w_j \mid j = 1,\ldots,n\}$ and let $W := \sum_{j=1}^{n} w_j$. Now the data for the corresponding instance I' of CARDINALITY (2-KP) is defined by

$$\begin{aligned} w_{1j} &:= w_j, & c_1 &:= W/2, \\ w_{2j} &:= w_{\max} - w_j, & c_2 &:= \tfrac{n}{2} w_{\max} - W/2, \\ K &:= \tfrac{n}{2}. \end{aligned}$$

Observe that for every subset of items S fulfilling (9.28) the same subset S from I' will fulfill all three conditions of CARDINALITY (2-KP) with equality.

Whenever a set S gives a positive answer to problem CARDINALITY (2-KP) for an instance I', there must be $|S| = n/2$. This can be seen by adding (9.30) to (9.31) which yields $|S| w_{\max} \leq \frac{n}{2} w_{\max}$. Together with (9.29) the claim follows. But with $|S| = n/2$ inequality (9.31) simply reads $\sum_{j\in S} w_j \geq W/2$ since $w_{\max}$ cancels out. Together with (9.30) it follows that $\sum_{j\in S} w_j = W/2$ which yields a positive answer to EQUIPARTITION.

Hence, also CARDINALITY (2-KP) is $\mathcal{NP}$-complete and its maximization version, where (9.29) is replaced by $\max |S|$, is $\mathcal{NP}$-hard. This is in fact a (2-KP) with $p_j = 1$ for $j = 1,\ldots,n$.

Finally, standard techniques can be employed to derive the statement of the theorem from the $\mathcal{NP}$-hardness of the cardinality maximization problem (cf. [164], Section 6.2). Indeed an *FPTAS* would have to produce an ε-approximate solution for any $\varepsilon > 0$ in time polynomial in n and $1/\varepsilon$. If we choose $\varepsilon := \frac{1}{n+1}$ we get for every ε-approximate solution with value z^A

$$z^A \geq (1-\varepsilon)z^* > z^* - z^*/n \geq z^* - 1, \tag{9.32}$$

where the last inequality follows from the trivial bound $z^* \leq n$ given by the definition of CARDINALITY (2-KP). The integrality of the solution values implies that

z^A must be equal to z^*. But this means that the existence of an *FPTAS* implies a polynomial time exact algorithm for this $\mathcal{NP}$-hard special case of (2-KP). □

A generalization of Theorem 9.4.1 was given by Magazine and Chern [312] who showed that the above statement holds also if the binary constraint (9.3) is replaced by arbitrary integer or infinite bounds on the variables. A different negative result about approximation for a minimization version of (d-KP) was given by Dobson [111].

9.4.2 Polynomial Time Approximation Schemes

The drawback of a *PTAS*, namely the exponential increase of the running time with respect to the desired accuracy, was already illustrated for (KP) in Section 6.1. However, in the light of Theorem 9.4.1 a *PTAS* is the only approximation scheme we may hope for.

It seems that the earliest *PTAS* was given by Chandra, Hirschberg and Wong [77] in 1976. At this time it was not known yet whether linear programs can be solved in polynomial time which made the direct inclusion of the LP-relaxation of (d-KP) in a *PTAS* impossible. Drawing from more recent results about the solution of special linear programs we will concentrate on a comprehensive description of the best available *PTAS* for d fixed.

The most natural idea for developing a *PTAS* is to follow the construction of the polynomial time approximation scheme H^ε for (KP) as given in Section 2.6. Basically, one only has to replace the subprocedure Ext-Greedy by an appropriate heuristic for (d-KP) and adjust the parameter ℓ indicating the cardinality of the enumerated subsets. The first scheme along these lines was given by Frieze and Clarke [156]. Its running time was reduced by a factor of n by Caprara, Kellerer, Pferschy and Pisinger [65]. Their algorithm is in principal identical to algorithm H^ε from Section 2.6, but instead of subprocedure Ext-Greedy the following $1/(d+1)$–approximation algorithm $H^{1/(d+1)}$ described in Figure 9.1 is plugged in.

Algorithm $H^{1/(d+1)}$:

compute an optimal basic solution x^{LP} of the
LP-relaxation of (d-KP) $\quad x^{LP}$ *has* $\leq d$ *fractional values*

$I := \{j \mid x_j^{LP} = 1\}$ *variables with integer values*

$F := \{j \mid 0 < x_j^{LP} < 1\}$ *variables with fractional values*

$z^H := \max\left\{\sum_{j\in I} p_j, \max\{p_j \mid j \in F\}\right\}$

Fig. 9.1. Greedy algorithm $H^{1/(d+1)}$ for (d-KP) with performance guarantee $1/(d+1)$.

Lemma 9.4.2 *$H^{1/(d+1)}$ is a $1/(d+1)$–approximation algorithm.*

Proof. The statement follows as a straightforward generalization of Theorem 2.5.4. Indeed, there is

$$z^* \leq z^{LP} \leq \sum_{j \in I} p_j + d\, F_{\max} \leq (d+1)\, z^H.$$

□

Choosing $\ell := \min\{\lceil d/\varepsilon \rceil - (d+1), n\}$ a *PTAS* for (d-KP) is obtained from H^ε after replacing Ext-Greedy by $H^{1/(d+1)}$ and removing before its execution any items exceeding the residual knapsack capacity in any of the dimensions (see [65]). The feasibility check for every subset L is replaced by the obvious analogon.

In order to show that the resulting algorithm is indeed a $(1-\varepsilon)$–approximation algorithm it is sufficient to follow the proof of Theorem 2.6.2. In particular, it must be shown that (in the notation of Theorem 2.6.2)

$$z^A \geq \sum_{j \in L^*} p_j + z_S^H \geq \frac{\ell+1}{\ell+d+1} z^* \geq (1-\varepsilon) z^*.$$

The two cases to be considered in the corresponding proof are given by $\sum_{j \in L^*} p_j \geq \frac{\ell}{\ell+d+1} z^*$ and $\sum_{j \in L^*} p_j < \frac{\ell}{\ell+d+1} z^*$. The argument is concluded by replacing the $1/2$-performance guarantee of Ext-Greedy by the $1/(d+1)$ ratio of $H^{1/(d+1)}$ shown in Lemma 9.4.2.

The running time of the resulting algorithm is dominated by the $O(n^\ell)$ executions of $H^{1/(d+1)}$ for each of the sets L with cardinality $|L| = \ell$. It was shown by Megiddo and Tamir [343] that the LP-relaxation of (d-KP) for d fixed can be solved in $O(n)$ time, which yields an overall running time of $O(n^{\ell+1})$. Clearly, the improvements of Section 6.1.1 cannot be applied to (d-KP). Hence, we can summarize the discussion by the following theorem from [65].

Theorem 9.4.3 *There exists a* PTAS *for (d-KP) with running time $O(n^{\lceil d/\varepsilon \rceil - d})$.*

Note that the *PTAS* for (KP) from Section 2.6 can be seen as a special case of the current construction with $d = 1$.

9.5 Heuristic Algorithms

The inherent computational difficulty of (d-KP) motivated the development of a considerable number of heuristic algorithms which compute feasible solutions of "reasonable quality" within "reasonable running time". Of course, it is difficult to

give accurate and well-founded comparisons between different heuristics. The usual approach to do so is the performance of computational experiments. However, these comparisons offer obvious drawbacks (differences in hardware and software environment, quality of implementation, generation of test data, etc.) which were addressed e.g. by Jackson et al. [247] and Barr et al. [27]. Since the published results of such experiments were derived over a period of several decades, the authors believe that it is of limited use to report them in detail.

In this section we will give an overview of several heuristics from the last 30 years. The more recent development of metaheuristics led to quite a number of contributions concerned with (d-KP). Hints to the corresponding literature will be given in Section 9.5.5.

All the heuristics are divided into three parts one of which is filled by greedy-type heuristics (cf. Caesar [60]). The second group is based on the solution of the LP-relaxation of (d-KP) whereas the more complex methods are loosely grouped together as a third cluster.

9.5.1 Greedy-Type Heuristics

The most obvious idea to find a feasible solution for (d-KP) is based on the same construction as algorithm Greedy for (KP) from Section 2.1. Also for (d-KP) we can consider the n items one after the other and put an item into the knapsack if it fits, i.e. if adding this item would not violate any of the d inequalities (9.2). Since we keep a feasible solution throughout the greedy process this approach is also called *primal greedy heuristic* in analogy to linear programming theory.

The so-called *dual greedy heuristic* starts by putting *all* items into the knapsack, i.e. by setting $x_1 = \ldots = x_n = 1$, thus generating an infeasible solution for all nontrivial instances.

Then we consider the items one after the other and remove each of them from the knapsack until the corresponding solutions is feasible. This would result in a solution with a threshold item t such that $x_1 = \ldots = x_t = 0$ and $x_{t+1} = \ldots = x_n = 1$.

Since removing the last item t may produce a large gap in the d constraints, a post-processing routine is usually applied which goes through the items 1 up to $t-1$ and tries to add them again into the knapsack. This means basically that a primal greedy heuristic is performed on the problem which remains after packing items $t+1$ until n.

While these two strategies are fairly simple and most authors seem not to deviate from this outline (see e.g. Fox and Scudder [150] for a general description) the crucial point, namely the order in which the items are considered, leaves plenty of room for discussion. The presence of d constraints voids any direct analogon to the ordering by decreasing efficiencies (2.2) for (KP). However, various models were developed in the literature to assign values expressing the expected "attractiveness"

of an item, sometimes called "bang-for-buck" ratio (e.g. Pirkul [378]). After fixing such efficiency values, the items are sorted in decreasing order for the primal greedy heuristic and in increasing order for the dual greedy heuristic.

A natural choice for the *efficiency* e_j of an item is the aggregation of all d constraints. In direct generalization of (2.2) Dobson [111] defined the following ratios e_j for every item j:

$$e_j := \frac{p_j}{\sum_{i=1}^{d} w_{ij}}. \tag{9.33}$$

An obvious drawback of this naive setting is the fact that different orders of magnitudes of the constraints is not considered. Hence, one constraint may completely dominate the ordering of the e_is with no inherent justification. An easy way to overcome this behaviour is the scaling of all inequalities which results in

$$e_j := \frac{p_j}{\sum_{i=1}^{d} \frac{w_{ij}}{c_i}}. \tag{9.34}$$

A slightly different way to take into account the relative contribution of the constraints was given by Senju and Toyoda [429] for the dual greedy heuristic. They compute the difference between the capacity and the total weight of all items in the corresponding dimension and define

$$e_j := \frac{p_j}{\sum_{i=1}^{d} w_{ij}(\sum_{j=1}^{n} w_{ij} - c_i)}. \tag{9.35}$$

Note that $e_j > 0$ for $j = 1,\ldots,n$, since any constraint i with $\sum_{j=1}^{n} w_{ij} \leq c_i$ could be removed from the problem instance in analogy to assumption (1.15). As in (9.34) scaling will be useful.

A more general framework to assign the appropriate "importance" to every constraint was discussed by Fox and Scudder [150]. Introducing *relevance* values r_i for every constraint $i = 1,\ldots,d$, one can define

$$e_j := \frac{p_j}{\sum_{i=1}^{d} r_i w_{ij}}. \tag{9.36}$$

The higher the relevance value of a constraint, the higher the "scarcity" of the corresponding resource and the less attractive it becomes to pack an item which consumes a lot of that resource. Trivially, (9.33) is attained for $r_1 = \ldots = r_d = 1$. Moreover, setting $r_i := 1/c_i$ resp. $r_i := (\sum_{j=1}^{n} w_{ij} - c_i)$ also (9.34) resp. (9.35) can be seen as special cases of (9.36). An interpretation of r_i as surrogate multipliers will be briefly considered in Section 9.5.2.

More sophisticated procedures adapt the relevance values during the execution of the greedy heuristic, i.e. after every decision on an item. An early version of this recomputation of relevances, resp. efficiencies, was discussed also by Senju and Toyoda [429].

Let X_j be the set of items packed into the knapsack after considering items $\{1,\dots,j\}$ by the primal greedy heuristic. In this context Fox and Scudder [150] proposed to set $r_i = 1$ for all constraints i where the remaining capacity $c_i - \sum_{k\in X_j} w_{ik}$ is minimal and $r_i = 0$ otherwise. Also scaling by c_i may be applied. In their paper special attention is given to the way of breaking ties and the importance to care about this frequently neglected detail is highlighted.

Based on a geometric interpretation Toyoda [461] gave a rather twisted description of the following construction in a primal greedy heuristic. We will give a slightly simplified version omitting a constant scaling factor thus loosing the geometric analogy.

Setting $r_i := \sum_{k\in X_j} w_{ik}$ in every iteration takes into account that throughout the execution resources get "scarcer" the more they are consumed. Moreover, it makes sense to scale the resource requirements by setting

$$r_i := \frac{1}{c_i} \sum_{k\in X_j} \frac{w_{ik}}{c_i}. \tag{9.37}$$

In the first iteration with $X_0 = \emptyset$ we use $r_i := 1/c_i$ instead. After every iteration of the primal greedy heuristic one should try to reduce the size of the remaining instance (cf. [461], [150]). This can be done by removing an item, if it is larger than the remaining capacity in one of the constraints, or removing a constraint, if it is fulfilled even by packing all remaining items. Note that this reduction may influence the ordering of the efficiencies.

In order to distinguish the crucial and most scarce resources more clearly from the not yet critical constraints Toyoda [461] suggested a refined choice (called "origin-moving" from a geometric perspective). It puts a higher relevance on resources which were already consumed to a high extent and a lower or zero relevance on those which are still in wider supply. In particular, Toyoda defined several models which can be unified by the following definition with constant parameters $t_1 \in [0,1[$ and $t_2 \in [1,3]$.

$$r_i := \max\left\{0, \frac{1}{c_i}\left(\sum_{k\in X_j} \frac{w_{ik}}{c_i} - t_1 \left(\max_{\ell=1}^{d}\left\{\sum_{k\in X_j} \frac{w_{\ell k}}{c_\ell}\right\}\right)^{t_2}\right)\right\}$$

This formula for r_i means that from the values in (9.37) a term depending on the relatively most consumed resource is subtracted and those resources with relatively low consumption are set to zero relevance. Note that the range of the parameters guarantees that at least the relevance of the constraint where the maximum is attained remains nonnegative.

Toyoda investigated e.g. the cases $t_1 = 0$, $t_1 = 0.2$ resp. $t_1 = 0.5$ for $t_2 = 1$ and for $t_1 = 1$ several choices of t_2, such as $t_2 = 2$ or taking the maximum over a whole

range of values for t_2. His computational experiments indicate the usefulness of employing the refined relevance values but do not exhibit a clear dominance among these choices.

An improvement and extension of the work of Toyoda for primal greedy heuristics was done by Loulou and Michaelides [307]. Given a *penalty value* v_j they defined for every item j as a formal generalization of (9.33) an efficiency e_j simply given as

$$e_j := \frac{p_j}{v_j}.$$

For simplicity of notation we assume that after every iteration a renumbering of items takes place. To find values for v_j which lead to a more successful application of the greedy heuristic than those resulting from the definition of relevance in (9.36) they identified three different quantities q_{ij} for every constraint i illustrating the potential effect of adding item j after items 1 to $j-1$ were processed.

$$\begin{aligned} q_{ij}^1 &:= \textstyle\sum_{k \in X_{j-1}} w_{ik} + w_{ij} && \text{consumed amount of resource } i, \\ q_{ij}^2 &:= c_i - \left(\textstyle\sum_{k \in X_{j-1}} w_{ik} + w_{ij} \right) && \text{remaining amount of resource } i, \\ q_{ij}^3 &:= \textstyle\sum_{k=j+1}^{n} w_{ik} && \text{maximal future demand for resource } i, \end{aligned}$$

where X_j still denotes the subset of items selected from $\{1, \ldots, j\}$. The penalty value is then derived by computing the product respectively quotient of the three quantities and taking the maximum over all resources after scaling.

$$v_j := \max_{i=1}^{d} \left\{ \frac{q_{ij}^1 \cdot q_{ij}^3}{c_i \cdot q_{ij}^2} \right\}$$

Loulou and Michaelides [307] give a number of remarks on fine-tuning this approach such as reducing the influence of q_{ij}^2 and q_{ij}^3 by taking the square root of these quantities or switching to a special procedure as soon as one of the resources is almost completely used up.

The extensive computational experiments in their paper suggest that different types of test instances will require a different setting of parameters but in general the approach delivers very reasonable solutions in a short time. It also exceeds the performance of Toyodas heuristics [461] with only a slight increase in computation time.

A more theoretical contribution by Rinnooy Kan, Stougie and Vercellis [407] studies the quality of the primal greedy heuristic solution as a function of the relevance values as applied in (9.36). Assigning to every item j a point in $\mathbb{R}^d$ with the ratios $(w_{1j}/p_j, \ldots, w_{dj}/p_j)$ as coordinates, the relevance values $(r_1, \ldots, r_d)$ can be seen as a normal vector of a hyperplane.

In this context, sorting by efficiencies corresponds to moving this hyperplane towards the points assigned to every item starting at the origin. Rinnooy Kan et

al. [407] give a nice connection between this geometric setup and the dual problem of the LP-relaxation of (d-KP). In particular, they can show that the relevance values derived from the optimal dual solution of the LP-relaxation yield a solution z^G of the primal greedy heuristic which contains as a subset the rounded-down solution of the optimal primal LP-relaxation of (d-KP). Together with Lemma 9.2.1 this yields immediately $z^G \leq z^* + d\, p_{\max}$ which cannot be improved upon (see Theorem 9.5.1).

These considerations lead to the obvious question for the *best* choice of relevance values, i.e. the computation of values r_j which yield the provably best primal greedy solution value. It is shown in [407] that this problem can be indeed solved for fixed values of d by an enumeration scheme based on the above geometric interpretation. However, this involves $O(n^2 \log n)$ executions of the greedy heuristic which will often be computationally unattractive. If $d > n$ holds, the same problem was even shown to be $\mathcal{NP}$-hard. Concerning the quality of this best primal greedy solution value the following (in some sense negative) result was given by Rinnooy Kan et al. [407].

Theorem 9.5.1 *The primal greedy algorithm for (d-KP) with optimal relevance values r_j has an* absolute *performance guarantee of $d \cdot p_{\max}$ and this bound is tight.*

However, it was also shown in [407] that for a particular probabilistic model there exist relevance values for which the resulting greedy solution value is asymptotically optimal with high probability (see Section 14.7).

While the greedy-type heuristics are easy to implement and in many applications "good enough" in the sense that one trusts the computed solution to be reasonably close to the unknown optimal solution, no relative performance guarantees were derived to affirm this believe.

To the best of our knowledge, beside the result of Theorem 9.5.1 there exists only one exception, namely a worst-case analysis by Dobson [111] based on the classical result of Chvatal [85]. Starting with a variant of (d-KP) with unbounded variables, i.e. (9.3) replaced by $x_j \in \mathbb{N}$, he defined a primal greedy heuristic based on (9.33). Denoting the partial harmonic series by $H(n) := \sum_{k=1}^{n} 1/k$ (note that $H(n)$ is of order $O(\ln(n))$) he showed in a very elaborate proof that the approximation ratio of the greedy heuristic is given by function H applied to the maximal cumulated resource requirements of one item. This result was also extended to the case where upper bounds are placed on the variables and hence also holds for the standard binary version of (d-KP).

Theorem 9.5.2 *The primal greedy algorithm for (d-KP) has a relative performance guarantee of $H\left(\max_{j=1}^{n}\left\{\sum_{i=1}^{d} w_{ij}\right\}\right)$ and this bound is tight.*

9.5.2 Relaxation-Based Heuristics

A major field of research on heuristics for (d-KP) deals with the utilization of relaxations as given in Section 9.2. Whereas the contents of the previous section on greedy-type heuristics were more or less tailor-made for (d-KP), many contributions to the current section apply also for more general integer programs. Hence, we only give a restricted overview of some ideas but do not cover this area extensively.

A fast and simple heuristic to derive a feasible solution x^H from successive LP-relaxations was given by Bertsimas and Demir [34] in the context of an approximate dynamic programming approach which requires a large number of heuristic solutions for different subproblems (see Section 9.5.4). Their *adaptive fixing heuristic* starts by solving the LP-relaxation in a first phase where it sets for a parameter $\gamma \in [0,1]$

$$x_j^H := \begin{cases} 1 & \text{if } x_j^{LP} = 1, \\ 0 & \text{if } x_j^{LP} < \gamma. \end{cases} \tag{9.38}$$

For the subproblem defined by the undecided variables x_j^H and the residual capacities a new LP-relaxation is solved. In this second phase further variables are fixed by setting

$$x_j^H := \begin{cases} 1 & \text{if } x_j^{LP} = 1, \\ 0 & \text{if } x_j^{LP} = 0, \\ 0 & \text{for } j = \arg\min\{x_j^{LP} \mid 0 < x_j^{LP} < 1\}. \end{cases} \tag{9.39}$$

The last assignment guarantees that at least one variable (the one with the smallest fractional value in the solution of the LP-relaxation) is set to 0 in every execution of Phase 2. This second phase is iterated for the resulting subproblems of decreasing size until all variables are resolved.

As may be expected, it turns out that small values of γ (i.e. $\gamma < 0.1$) yield better solution values while consuming more running time. Since fast solutions were required in the application in [34] the authors set $\gamma = 0.25$. Note that the case $\gamma = 1$ is equivalent to computing the rounded down solution of the LP-relaxation. Computational experiments reported by Bertsimas and Demir [34] exhibit the dominance of this heuristic over all greedy-type heuristics from Section 9.5.1.

Turning to more complicated approaches, one of the oldest heuristics in this domain dating back to 1969 is due to Hillier [228]. His approach, which was formulated for the unbounded case, consists of three phases. In Phase 1 the optimal solution x^{LP} of the LP-relaxation of (d-KP) and a second solution are computed. The latter should have the property of being "close" to the former, it should be feasible for the LP-relaxation and it should be possible to round this solution to a feasible binary solution of (d-KP). Strategies to find this crucial second solution based on linear programming theory were elaborated in [228].

In Phase 2 a search procedure moves along a straight line segment from x^{LP} towards the second solution and looks for a feasible binary solution "near" to this line. Due to the properties required from the second solution this search is bound to be successful. Finally, in Phase 3 a local improvement procedure is applied. It consists of exchanging systematically the values of one or two variables, i.e. change x_j to $1-x_j$, thus looking for a new feasible and possibly better solution. A computational comparison of this heuristic with the method by Senju and Toyoda [429] and a more general algorithm by Kochenberger et al. [280] was reported by Zanakis [496]. He goes into a lot of details and statistical evaluations to explain the particular behaviour of these three heuristics for many different types of instances.

A widely cited heuristic method for general 0–1 programming was given by Balas and Martin [20] in 1980. The main idea of their approach is based on the fact that in any basic feasible solution of the LP-relaxation of (d-KP) there are n nonbasic variables (cf. the proof of Lemma 9.2.1). If these would all correspond to original variables $x_1,\ldots,x_n$ or to slack variables denoting the gap $1-x_j$, we would have an integer solution. Therefore, it would be a good idea to have as basic variables as many slack variables y_i as possible corresponding to the constraints (9.2).

The procedure by Balas and Martin starts by solving the LP-relaxation and continues to perform pivot operations which either brings an additional slack variable y_i into the basis or which exchanges two slack variables thus leaving the distribution of basic and nonbasic variables unchanged but reducing the "deviation from integrality". This deviation is simply measured by summing up the values $\min\{x_j, 1-x_j\}$ over all basic non-slack variables x_j. If no feasible integer solution can be found through rounding respectively truncation after this phase of pivot operations, slack variables can also be forced into the basis by a pivot operation which abandons feasibility.

The remainder of the procedure consists mainly of so-called *complement operations*. This means that not unlike Phase 3 in the heuristic by Hillier [228] a set of one, two or three variables is complemented in an attempt to reach better solutions from a given feasible solution or to get a feasible integer solution if the previous pivot operations were unsuccessful. This approach can be seen as a direct predecessor of modern *neighborhood* structures which are a crucial element of metaheuristics (cf. Section 9.5.5). For more algorithmic details we refer to [20].

An interesting algorithm which incorporates the features of Lagrangian multipliers (cf. Section 9.2) into the dual greedy heuristic was developed by Magazine and Oguz [315]. Let X be the set of items currently packed into the knapsack. As in the dual greedy heuristic their approach starts by including all items into the knapsack ($X := N$) and successively removes items until a feasible solution is found.

The criterion for the removal of items works in two steps. In the first step a "most violated" constraint i^* is selected. Therefore the total *scaled demand* respectively consumption of every resource i is computed as $d_i := \sum_{j\in X} \frac{w_{ij}}{c_i}$. The constraint is then determined by

$$i^* := \arg\max\{d_i \mid i = 1,\ldots,d\}. \tag{9.40}$$

In the second step the item to be removed is identified. Therefore, relative costs are kept for every item j following a Lagrangian relaxation approach. A set of multipliers $\lambda_1,\ldots,\lambda_d$ (all of them initialized to 0) is kept with λ_i assigned to the ith constraint in (9.2). The relative costs of item j are given by $p_j - \sum_{i=1}^{d} \lambda_i \frac{w_{ij}}{c_i}$. Now we try for each item j separately to increase the multiplier of i^* to $\lambda_{i^*} + \delta_j$ with δ_j as large as possible such that the relative costs of item j remain nonnegative. This yields

$$p_j - \sum_{i=1}^{d} \lambda_i \frac{w_{ij}}{c_i} - \delta_j \frac{w_{i^*j}}{c_{i^*}} = 0. \tag{9.41}$$

So we can deduce

$$\delta_j = \frac{p_j - \sum_{i=1}^{d} \lambda_i \frac{w_{ij}}{c_i}}{\frac{w_{i^*j}}{c_{i^*}}}. \tag{9.42}$$

Now the variable x_{j^*} determined by

$$j^* := \arg\min\{\delta_j \mid j \in X\}$$

is changed from 1 to 0 and item j^* eliminated from X. The multiplier is updated by $\lambda_{i^*} := \lambda_{i^*} + \delta_{j^*}$. The update scheme for the multipliers λ_i was improved by Volgenant and Zoon [477] who modify more than one multiplier in every iteration.

This procedure is continued until a feasible solution is found, i.e. $d_i \leq 1$ for all $i = 1,\ldots,d$. As mentioned in Section 9.5.1 such a dual greedy-type heuristic can be improved by checking whether any removed items can be reinserted into the knapsack after the first feasible solution is found. A theoretical interpretation of this heuristic and its connection to Lagrangian relaxation can be found in Magazine and Oguz [315] together with a number of computational experiments.

Note that a particular advantage of this method is the availability of upper bounds on the optimal solution value which facilitates a more meaningful judgement of the quality of the attained approximate solution. This upper bound, which is present without solving any LP, is given by

$$z^* \leq \sum_{j \in X} p_j + \sum_{i=1}^{d} (1 - d_i)\lambda_i. \tag{9.43}$$

Recall that X denotes the set of currently packed items. In the paper of Volgenant and Zoon [477] a more sophisticated computation of multipliers is described which yields a better (i.e. lower) upper bound at a modest increase of running time.

A second theoretical result in [315] related to the reduction in Section 9.2 permits to definitely determine the optimal values of some variables. If the relative cost of some variable x_j is larger than the gap between the heuristic solution value and the upper bound (9.43), i.e. if

$$\left| p_j - \sum_{i=1}^{d} \lambda_i \frac{w_{ij}}{c_i} \right| > \sum_{i=1}^{d} (1 - d_i)\lambda_i,$$

then this variable must keep its value computed by the heuristic also in every optimal solution.

The definition of relevance values r_i for every constraint in Section 9.5.1 can be also seen as kind of surrogate multipliers as introduced in Section 9.2. A primal greedy heuristic based on this notion of surrogate relaxation was given by Pirkul [378]. Indeed, for a set of d surrogate multipliers $\mu_1, \ldots, \mu_d$ the relaxation of (d-KP) is an instance of (KP) and the classical definition of efficiencies for (KP) can be applied yielding the same structure as (9.36),

$$e_j := \frac{p_j}{\sum_{i=1}^{d} \mu_i w_{ij}}. \tag{9.44}$$

The main point of Pirkul's heuristic is the computation of "good" surrogate multipliers for μ_i by a procedure based on the LP-relaxation of single constraint knapsack problems (see [378] for details).

After running the primal greedy heuristic for these multipliers a local improvement phase along the same lines as in [228] is performed. For every variable which was set to 1 the reduced instance resulting from setting this variable to 0 is solved again by the greedy heuristic and the best attained solution value is reported.

Computational experiments suggest that this improvement step yields a major gain in quality of the solution value while the running time is only mildly effected since most of the computational effort is spent on finding the surrogate multipliers. Comparisons to the method by Loulou and Michaelides [307] indicate that Pirkul's heuristic yields better solutions at the cost of a slightly higher running time. Compared to the algorithm by Balas and Martin [20] the present heuristic turned out to be faster while the solution quality could not be clearly distinguished.

9.5.3 Advanced Heuristics

More recent contributions to the area of (d-KP)–heuristics draw from the experiences described in the preceding sections and try to exploit the successful features of different approaches. The resulting *hybrid* algorithms require more running time than straightforward greedy heuristics but through a clever combination of variable reduction, relaxations and multipliers respectively relevance values they attain very good solution values close to the optimum for many classes of test instances. Clearly, the construction of such algorithms requires careful fine-tuning and often depends on one or more parameters which steer the execution towards a reasonable trade-off between running time and solution quality (i.e. deviation from the optimal solution value).

A typical approach of this flavour is due to Lee and Guignard [296]. Their heuristic applies a modification of the primal greedy heuristic by Toyoda [461]. Instead of deciding on one item separately in every iteration, the modified version selects

a larger set of items at once before recomputing the relevance values thus saving computation time (but possibly loosing solution quality).

Starting from the feasible solution resulting from this modified heuristic the LP-relaxation of (d-KP) is computed. Guided by a parameter the reduced costs of the optimal LP-solution are used together with a comparison to the initial solution computed by the modified Toyoda heuristic to reduce the problem size and fix some of the variables. This approach is iterated which makes the faster performance of the modified Toyoda heuristic especially valuable. At the end of each iteration the complement operations of the Balas and Martin [20] algorithm are performed controlled by a second parameter. The overall iteration number is again dependent on a third parameter. For more algorithmic details the reader is referred to [296].

Extensive computational experiments give hints on the selection of the three parameters which are clearly crucial for attaining the desired compromise between running time and deviation from optimality. This also provides for some flexibility to meet the requirements posed by a real-world application. Some comparisons to other heuristics suggest that the procedure produces better results than the methods of Toyoda [461] and Magazine and Oguz [315] while consuming less running time than the latter but not the former. It seems to be faster than the algorithm by Balas and Martin [20] but reaches slightly worse solution values.

Advancements in the application of surrogate multipliers extending and improving the paper by Pirkul [378] were given in the work of Fréville and Plateau [154] (and its predecessor [152]) mentioned earlier in Section 9.2. The authors present a four phase heuristic called AGNES which can be briefly outlined as follows.

Phase 1 performs a primal greedy algorithm with an ordering given by the efficiencies according to (9.44) for some initial multipliers μ_i, $i = 1,\ldots,d$. In Phase 2 the linear relaxation of the surrogate relaxation defined by the multipliers μ_i is solved. This is simply an LP-relaxation of a (KP) instance. However a slightly perturbed right hand side is used given by $(1+\alpha)\sum_{i=1}^{d}\mu_i c_i$ for a small parameter α. All variables which are 1 in the solution of this "double" relaxation are for the time being set to 1 also for (d-KP). If the resulting solution is infeasible for (d-KP) a dual greedy procedure is applied to find a feasible solution. Now the procedure is repeated recursively for the remaining subproblem consisting of the variables which are 0 in the computed feasible solution and the residual capacities. In this way several feasible solutions are computed for different values of α. Fréville and Plateau suggest to choose $\alpha \in [-0.2, 0.2]$.

A different scheme to fix the value of a certain number of variables is undertaken in Phase 3. Here the reduced costs of the solution to the above relaxation of the surrogate problem are taken into account. For a parameter $\beta \in [1/5, 1/3]$, the βn variables with the highest resp. lowest reduced costs are set to 1 resp. 0. However, the setting to 1 is only done as long as feasibility is preserved. As in Phase 2 this process is performed recursively for the remaining subproblem and the whole process iterated for different values of β yielding a set of feasible solutions.

In the final Phase 4 an interchange heuristic similar to the complement operations by Balas and Martin [20] is performed. To keep the running time low it is suggested in [154] to use only exchange steps involving one variable at a time.

The whole heuristic AGNES is executed several times for different sets of multipliers μ_i. These are computed e.g. as optimal dual solution of the LP-relaxation of (d-KP), as optimal solution of one of the dual problems defined in (9.16) but with relaxed variables $x_j \in [0,1]$, or simply as

$$\mu_i := \frac{\sum_{j=1}^{n} w_{ij} - c_i}{\sum_{j=1}^{n} w_{ij}}.$$

Clearly, the best feasible solution encountered during the overall computation is kept as final result.

Some computational experiments are reported in [154]. They clearly indicate that the presented heuristic produces better solutions than the algorithm by Pirkul [378], but requires a considerable amount of extra running time. The reported solutions are also at least as good as those from Balas and Martin [20] but no plausible comparison of running times is available. For many medium-sized test instances from the literature the heuristic was shown to find an optimal solution.

Summarizing, it can be said in general that advanced heuristics pick from classical elements such as greedy algorithms and various relaxations and go through a series of combinations between them. Depending on the values of certain control parameters the heuristics can either exploit modern computing power and reach near-optimal solution after some time or stick to very short running times at the risk of a larger deviation from the optimal solution value.

9.5.4 Approximate Dynamic Programming

It was pointed out in Section 9.3.2 that the effort for classical dynamic programming explodes with the number of constraints. However, if we drop the objective of finding an optimal solution an approximate algorithm can be deduced from dynamic programming. The resulting *approximate dynamic programming* approach is due to Bertsimas and Demir [34]. We will outline the basic idea of their algorithm and briefly mention some of the technical details and improvements which are crucial for the practical performance of the method. For a full description we refer to [34].

After computing the dynamic programming function values $z(j, g_1, \ldots, g_d)$ from the recursion (9.26) the optimal solution x^* can be determined by moving *backwards* from x_n^* down to x_1^*. This means that we set $x_n^* = 1$ if

$$p_n + z(n-1, c_1 - w_{1n}, \ldots, c_d - w_{dn}) > z(n-1, c_1, \ldots, c_d) \qquad (9.45)$$

and $x_n^* = 0$ otherwise. The analogous continuation of this procedure to determine $x_{n-1}^*, \ldots, x_1^*$ is obvious. In a suboptimal algorithm we can replace the costly computation of the optimal function values appearing in (9.45) by an upper bound and a heuristic solution yielding a lower bound $z^\ell(j, g_1, \ldots, g_d) \leq z(j, g_1, \ldots, g_d) \leq U(j, g_1, \ldots, g_d)$. Bertsimas and Demir [34] suggest the following approach to determine a heuristic solution x^A in an approximate dynamic programming approach. Since the upper and lower bounds for the two function values in (9.45) will have different relative deviations, an approximation with the same percentage deviation from the corresponding upper bound is reached by computing the "better" approximation ratio

$$k := \max\{z^\ell(n-1, c_1, \ldots, c_d)/U(n-1, c_1, \ldots, c_d),$$
$$z^\ell(n-1, c_1 - w_{1n}, \ldots, c_d - w_{dn})/U(n-1, c_1 - w_{1n}, \ldots, c_d - w_{dn})\}.$$

Then both upper bounds are "discounted" by the same approximation ratio k. If

$$p_n + kU(n-1, c_1 - w_{1n}, \ldots, c_d - w_{dn}) > kU(n-1, c_1, \ldots, c_d)$$

then $x_n^A = 1$, otherwise $x_n^A = 0$. Note that for one of the two expressions there is $z^\ell = kU$. Again, the procedure is easily extended for $x_{n-1}^A, \ldots, x_1^A$.

It should be pointed out that this approach can also be seen as a partial branch-and-bound procedure. Whereas in an exact branch-and-bound approach the complete tree is implicitly enumerated, the current heuristic algorithm pursues only a single path from the root node ($x_n^A = 0, 1$) to a leaf ($x_1^A = 0, 1$). In every node the decision of setting the corresponding variable equal to 0 or 1 is made once for all depending on the upper and lower bounds of the two corresponding subinstances.

Many details of approximate dynamic programming remain to be addressed. Clearly, a crucial point is the computation of the upper and lower bounds, which must be carried out roughly $2n$ times. Hence, the corresponding bounding procedures should be fast and possibly employ some restart feature since they are performed iteratively on closely related subinstances. In [34] the LP-relaxation (see Section 9.2) is used to compute the upper bound $U := \lfloor z^{LP} \rfloor$ and the adaptive fixing heuristic introduced at the beginning of Section 9.5.2 for z^ℓ.

As an alternative to the explicit bound computation an interesting estimation procedure for $z(j, g_1, \ldots, g_d)$ is presented by Bertsimas and Demir [34]. Based on theoretical results for randomly generated instances of (d-KP) (see Section 14.7) the authors indicate a certain linear dependence of the optimal solution value on the capacity vector. Hence, they propose to compute exact or approximate values only for some of the entries of z, namely for an appropriately chosen *sample set* of capacity vectors. The remaining entries are derived from these sample values by a weighted interpolation. Unfortunately, this interesting approach turns out to be dominated from a computational point of view by the simple adaptive fixing heuristic.

A considerable improvement of the performance of approximate dynamic programming is reached by fixing variables whenever possible by a reduction procedure (see

Sections 3.2 and 9.2) making use of the reduced costs of the LP-relaxation. Furthermore, early termination of the algorithm can be sometimes achieved by a criterion which identifies the "inadvertent" detection of an optimal solution during the execution. A full description of the technical details of this method can be found in [34]. The computational experiments in that paper point out that the resulting algorithm competes successfully with other heuristic approaches.

9.5.5 Metaheuristics

It was mentioned in the Introduction 9.1 that due to its computational difficulty (d-KP) is a popular subject for developing and testing modern metaheuristics. During the last ten years a sizeable number of papers have appeared which tackle in a competitive spirit more or less the same test instances by tabu search heuristics, genetic algorithms, evolutionary algorithms and more exotic heuristic concepts. Battiti and Tecchiolli [28] underline the importance of (d-KP) as a benchmark problem in their study on reactive tabu search. However, a self-contained treatment of all relevant metaheuristics would require an introduction into the basics of the corresponding algorithmic frameworks which are not specific for knapsack problems and go beyond the scope of this book. Hence, we will restrict ourselves to give some pointers to surveys and most recent papers.

A very good survey of existing contributions to this field was given by Hanafi, Fréville and El Abdellaoui [212]. The authors give pseudocode descriptions of the core routines required for different heuristics, discuss a wide range of references and also include some computational experiments.

The paper by Chu and Beasley [83] not only presents a new genetic algorithm for (d-KP) but also gives a concise but comprehensive survey of existing heuristics far beyond genetic algorithms. Very successful genetic algorithms were developed by Haul and Voss [216] and Raidl [401].

Tabu search turned out to be the method of choice for tackling combinatorial optimization problems in the last decade (cf. Glover and Laguna [184]). A tabu search algorithm for (d-KP) utilizing information from surrogate constraints was developed by Glover and Kochenberger [182]. A very well structured overview of tabu search approaches is given in the contribution by Hanafi and Fréville [211] who also develop a new sophisticated tabu search algorithm. Probably the most recent reference is due to Vasquez and Hao [471]. Their paper also reviews some of the most interesting developments of the last few years.

Elaborations on evolutionary algorithms for (d-KP) including references to related work can be found e.g. in the articles by Gottlieb [192, 191]. Finally, we would also like to mention the early application of simulated annealing to (d-KP) performed by Drexl [113] back in 1988.

9.6 The Two-Dimensional Knapsack Problem

Most instances of (d-KP) encountered in practice consist only of a relatively small number of constraints with $d < 10$ or even $d < 5$. Since (d-KP) is computationally very difficult to handle even for a moderate number of constraints, it seems to be reasonable to pay attention to the special case of the $d = 2$. In the Introduction 9.1 the cardinality constrained knapsack problem was mentioned. This can be seen as a very special case of (2-KP), sometimes interpreted as 1.5–*dimensional knapsack problem*, and will be handled in Section 9.7.

The *bidimensional knapsack problem* (2-KP) is an obvious candidate for detailed investigations, since it is already a "very hard" problem, in the sense that no *FPTAS* is likely to exist (see Theorem 9.4.1), but it still offers much better chances for the development of successful algorithms than the general (d-KP). Therefore, it seems rather surprising that to our knowledge no specific study was devoted to (2-KP) before the papers of Fréville and Plateau in the nineties of the last century.

The computation of bounds by relaxation methods (see Section 9.2) is clearly promising for (2-KP) since the relaxed problems usually turn out to be instances of (KP) or even linear relaxations of (KP). In particular, for the Lagrangian relaxation of (2-KP) usually a single multiplier $\lambda \geq 0$ is introduced to relax one of the constraints yielding the following problem. (We omit $M := \{2\}$ in the notation.)

$$\begin{aligned} z(L(\lambda)) = \text{ maximize } & \sum_{j=1}^{n} (p_j - \lambda w_{2j})\, x_j + \lambda c_2 \\ \text{subject to } & \sum_{j=1}^{n} w_{1j} x_j \leq c_1\,, \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned} \tag{9.46}$$

Computing $z(L(\lambda))$ for a fixed multiplier λ is equivalent to solving a standard knapsack problem. The corresponding Lagrangian dual problem defined by

$$z(LD) := \min\{z(L(\lambda)) \mid \lambda \geq 0\}$$

as in (9.16) can be solved as before by classical subgradient optimization. Thiongane et al. [457] present an improved subgradient algorithm for $z(LD)$ based on geometric properties of $z(L(\lambda))$ which is a convex function in λ. Furthermore, they add elements of a local search heuristic to speed up the subgradient algorithm. The instances of (KP) which have to be solved while searching for the optimal multiplier all have the same constraint and differ only in the objective function. This general property can be exploited by appropriate reoptimization techniques for the present case of subproblem (KP). Computational experiments in [457] illustrate the effectiveness of these techniques.

Classical results for Lagrangian relaxation, Lagrangian decomposition (cf. Section 3.9) and surrogate relaxation in the context of (2-KP) are stated by Fréville and Plateau [153]. The main part of their contribution concentrates on the solution of the surrogate dual problem. For a surrogate multiplier $\mu \in \mathbb{R}^2_+$ the surrogate relaxation of (2-KP) with optimal solution value $z(S(\mu))$ is given as in (9.8), (9.9) by

$$\begin{aligned} z(S(\mu)) = \quad & \text{maximize} \sum_{j=1}^{n} p_j x_j \qquad (9.47) \\ \text{subject to} \quad & \mu_1 \left(\sum_{j=1}^{n} w_{1j} x_j \right) + \mu_2 \left(\sum_{j=1}^{n} w_{2j} x_j \right) \leq \mu_1 c_1 + \mu_2 c_2 , \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,n, \end{aligned}$$

which is an instance of (KP) for any fixed μ. Since only $M := \{1,2\}$ makes sense, we omit M in the notation. The surrogate dual problem follows immediately as in (9.16),

$$z(SD) := \min\{z(S(\mu)) \mid \mu \in \mathbb{R}^2_+\}.$$

As outlined by Gavish and Pirkul [166], who used (2-KP) as a subroutine for the computation of surrogate multipliers for (d-KP) (see Section 9.2), w.l.o.g. μ_1 can be set to 1, due to scaling.

An advanced search procedure, where μ_2 can be deduced from a limited search interval of $[0,1]$, is given by Fréville and Plateau [153]. The effectiveness of their procedure SADE is shown by a series of computational experiments. The number of iterations, respectively computations of $z(S(\mu))$, by an exact algorithm for (KP) is quite stable in all reported tests.

An algorithm for computing optimal solutions for (2-KP) was presented by Fréville and Plateau [155] in 1996. Their approach is a framework incorporating several tools described in the preceding sections, in particular the solution of the surrogate dual problem by the above-mentioned method.

A rough description of the two phases of their algorithm runs as follows (for more details see [155]). In Phase 1 one tries to find an optimal solution through a series of iterations. In an inner loop of Phase 1 attempts are made to reduce the problem size as much as possible. To this end the two constraints are checked for redundancy and other straightforward tests for optimality are performed. Then the surrogate dual problem is solved providing an upper bound on the optimal solution value and the heuristic AGNES described in Section 9.5.3 is performed to attain also a lower bound. The results of these two subprocedures are used for the possible fixing of variables and reduction of coefficients. This inner loop is iterated until no further fixing respectively reduction is possible.

The outer loop of Phase 1 consists of a computationally more expensive subprocedure for fixing variables. Based on an idea by Gavish and Pirkul [166] one attempts

to fix variables to the value they attain in the optimal surrogate dual solution. If this works out, the inner loop is started again, otherwise Phase 1 is terminated.

Phase 2 consists of a branch-and-bound enumeration scheme making use of the bounds derived in Phase 1. Without giving many details the authors suggest a depth-first search strategy with an ordering of the nodes depending on the surrogate dual solution. It was observed that even imposing a low bound on the number of enumeration steps will still generate a near optimal solution for most problem instances. Computational experiments show the superiority of this method towards the algorithm by Gavish and Pirkul [166] for (d-KP).

Hill and Reilly [227] performed an extensive empirical study to investigate the effects of correlation between profits and the two weight coordinates on the performance of the heuristic by Toyoda [461] and the commercial integer programming solver CPLEX (see `www.ilog.com`). Not surprisingly, they found a significant influence of the correlation structure of the input data on the performance of algorithms. This influence turned out to be dissimilar for different solution methods, which means that it is hardly possibly to classify a problem instance a priori as "easy" or "difficult".

9.7 The Cardinality Constrained Knapsack Problem

In many practical occurrences of knapsack problems additional requirements are imposed on the structure of the optimal solution. A quite frequent and natural restriction concerns the number of items included in an optimal solution. Whenever the selection of items causes the explicit handling of goods or induces transaction costs not represented in the data of the knapsack instance, solutions with a small number of larger items will be preferred to a solution with a large number of smaller items. Introducing $\sum_{j=1}^{n} x_j$ explicitly as a second objective function to be minimized leads to a two-objective knapsack problem which opens up a range of further questions, such as solution concepts, etc. Some remarks on these multi-objective knapsack problems will be given in Section 13.1.

The easiest way of representing this goal is the formulation of a *cardinality constraint*, i.e. an upper bound k on the number of packed items. The resulting *cardinality constrained knapsack problem* or *k-item knapsack problem* (kKP) is given by

$$(k\text{KP}) \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{9.48}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \tag{9.49}$$

$$\sum_{j=1}^{n} x_j \leq k, \tag{9.50}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n, \tag{9.51}$$

where k is an integer number between 2 and $n-1$. Clearly, constraint (9.50) guarantees that there are at most k items packed into the knapsack.

Although cardinality constraints were already considered in the early work by Gilmore and Gomory [175], to our knowledge this "1.5–dimensional knapsack problem" was formally introduced by Caprara, Kellerer, Pferschy and Pisinger in [65]. In this section we will briefly review the main contents of their paper.

Cardinality constraints were also used to strengthen the LP-relaxation and Lagrangian relaxation of (KP) as described in Section 5.1.1. A continuous knapsack problem with a cardinality constraint was treated by de Farias and Nemhauser [101]. In their version the variables x_j can take an real value in $[0,1]$ but at most k of them are allowed to be different from 0. It is shown in [101] that with this constraint also the continuous knapsack problem is $\mathcal{NP}$-hard. In their paper a branch-and-cut framework is applied and different families of facet-defining inequalities are derived (cf. Section 3.10).

9.7.1 Related Problems

A problem closely related to (kKP) is the *exact k-item knapsack problem* (E-kKP) where the number of items in a feasible solution must be *exactly equal* to k. The mathematical programming formulation of (E-kKP) can be immediately derived from (kKP) by replacing (9.50) by

$$\sum_{j=1}^{n} x_j = k. \tag{9.52}$$

Note that (E-kKP) may not even have a feasible solution. However, this can be checked easily by comparing capacity c to the total weight of the k items with smallest weight.

Observe that (kKP) and (E-kKP) can be easily transformed into each other. More precisely, any instance of (kKP) can be solved as the following (E-kKP). The value of k remains the same, whereas the profits and the weights of the original items are multiplied by k and the new capacity is defined as $(k+1)c-1$, where c is the original capacity. Finally, k "dummy" items with profit and weight 1 are added. By definition, any subset of items with cardinality $k' \leq k$ is feasible for the original (kKP) instance if and only if it satisfies the knapsack constraint for the (E-kKP) instance. To satisfy also the cardinality equation (9.52), $k-k'$ dummy items are added. The ranking by profit of the feasible solutions of the two instances is the same due to profit scaling.

On the other hand, (E-kKP) can be solved as a (kKP) in the following way. Setting as before $P := \sum_{j=1}^{n} p_j$ and $W := \sum_{j=1}^{n} w_j$ there appears for every item j of (E-kKP) an item with profit $P + p_j$ and weight $W + w_j$ in the (kKP) instance. The capacity of the (kKP) instance is set to $kW + c$. Clearly, any feasible set of items in (E-kKP) trivially corresponds to a feasible solution of the (kKP) instance. Moreover, any feasible solution of (kKP) with at most $k-1$ items has a profit less than $(k-1)P+P$, whereas any feasible solution with k items has a profit greater than kP. Therefore, the (kKP) instance yields the same optimal solution as (E-kKP). An infeasible instance of (E-kKP) leads an objective function value less than kP.

Note that in [65] the concept of a cardinality constraint is also applied to the subset sum problem (SSP) thus defining the *k-item subset sum problem* (kSSP) and the *exact k-item subset sum problem* (E-kSSP) as the obvious special cases of (kKP) and (E-kKP). Clearly, the above transformations (the second one different from [65]) also apply to the subset sum variants.

9.7.2 Branch-and-Bound

A simple branch-and-bound algorithm for the unbounded version of (kKP) with (9.51) replaced by $x_j \in \mathbb{N}_0$ was presented by Zak and Dereksdottir [495]. Improving upon the classical approach by Gilmore and Gomory [175], they sort the items similar to (9.44) according to decreasing values of

$$e_j := \frac{p_j}{\mu_1 w_j + \mu_2}, \tag{9.53}$$

where μ_1 and μ_2 are chosen as the optimal dual solution of the LP-relaxation of (kKP).

Two bounds were applied in their branch-and-bound approach. The classical bound $U_1 = U_{LP}$ derived from solving the LP-relaxation of (kKP) yields quite good bounds but consumes significant running time. The following simple bound U_2 tries to fill the residual capacity of the knapsack with maximum profit using only one item type from the set of the remaining free variables F.

$$U_2 := \sum_{\ell \in N \setminus F} p_\ell x_\ell + \min \left\{ \max_{j \in F} \frac{p_j}{w_j} \left(c - \sum_{\ell \in N \setminus F} w_\ell x_\ell \right), \max_{j \in F} p_j \left(k - \sum_{\ell \in N \setminus F} x_\ell \right) \right\} \tag{9.54}$$

It can be evaluated much faster but does not match the quality of U_{LP}. More details about the computational experiments can be found in [495].

9.7.3 Dynamic Programming

In the following we will present dynamic programming algorithms for the cardinality constrained problems. The dynamic programming procedure for (kKP) is an

immediate adaptation of the schemes for (KP). Since the presented scheme is used subsequently to derive an *FPTAS*, we concentrate on dynamic programming by profits (cf. Section 2.3). However, it should be straightforward to exchange the role of profits and weights and derive an algorithm performing dynamic programming by weights.

Note that dynamic programming algorithms for (KP) with cardinality constraints are also considered by Pisinger [384] in an exact algorithm for a special class of (KP) instances and by Pferschy et al. [376] for the collapsing knapsack problem (see Section 13.3.7).

Let U be an upper bound on the optimal solution value of (kKP) computed e.g. by taking twice the approximate solution value of heuristic LP-Approx described in the following Section 9.7.4.

Sticking to the notation of DP-Profits we define the two-dimensional dynamic programming function $y_j(q,\ell)$ for $j = 1,\dots,n$, $\ell = 1,\dots,k$, $q = 0,\dots,U$, as the optimal solution value of the following problem:

$$y_j(q,\ell) = \min\left\{ \sum_{i=1}^{j} w_i x_i \;\middle|\; \sum_{i=1}^{j} p_i x_i = q,\ \sum_{i=1}^{j} x_i = \ell, x_i \in \{0,1\} \right\}$$

An entry $y_j(q,\ell) = w$ means that considering only the items $1,\dots,j$ there exists a subset of exactly ℓ of these items with total profit q and minimal weight w among all such subsets.

The initialization is given by $y_0(q,\ell) := c+1$ for $\ell = 0,\dots,k$, $q = 0,\dots,U$, and $y_0(0,0) := 0$. Then, for $j = 1,\dots,n$ the entries of y_j can be computed from those of y_{j-1} by the obvious extension of recursion (2.10)

$$y_j(q,\ell) = \begin{cases} y_{j-1}(q,\ell) & \text{if } q < p_j, \\ \min\{y_{j-1}(q,\ell), y_{j-1}(q-p_j,\ell-1)+w_j\} & \text{if } \ell > 0, q \geq p_j. \end{cases} \tag{9.55}$$

After computing all values of y from y_1 up to y_n the optimal solution value of (kKP) is given by $\max\{q \mid y_n(q,\ell) \leq c, q = 0,\dots,U, \ell = 0,\dots,k\}$, whereas the optimal solution value of (E-kKP) is given by $\max\{q \mid y_n(q,k) \leq c, q = 0,\dots,U\}$.

Determining the optimal solution of (kKP) (resp. (E-kKP)) by going through all necessary iterations of (9.55) requires $O(nkU)$ operations. In a straightforward implementation each such operation may require the manipulation of a set of up to k items. However, by introducing a pointer structure for the subsets of items similar to the construction in Section 2.3 every such operation can be done in constant time. More details about the implementation of this aspect are given in [65].

The storage requirements are given by $O(k^2U)$. Note that for the computation of y_j only the values of y_{j-1} have to be stored. Each of the kU entries of y_j may correspond to a subset of k items (also in an advanced representation) which brings

the total space requirements to $O(k^2U)$. However, the storage reduction scheme of Section 3.3 can be applied to save the explicit storage of the subsets thus reducing the space bound by a factor of k. Altogether this shows the following statement.

Theorem 9.7.1 *Problems (kKP) and (E-kKP) can be solved to optimality by dynamic programming in $O(nkU)$ time and $O(n+kU)$ space.*

For (kSSP) (resp. (E-kSSP)) a different dynamic programming scheme can be used. In contrary to all previous schemes in this book it does not compute weight or profit values but *counts* the number of items required to reach any given weight value.

For $j = 1,\ldots,n$ and $d = 0,\ldots,c$ let $g_j(d)$ be the optimal solution value of the following subinstance of (kSSP) defined on the first j items and a knapsack capacity d:

$$g_j(d) = \min\left\{\ell \;\middle|\; \sum_{i=1}^{j} w_i x_i = d,\ \sum_{i=1}^{j} x_i = \ell, x_i \in \{0,1\}\right\} \tag{9.56}$$

where $g_j(d) := n+1$ if no solution satisfies the two constraints. One initially sets $g_0(0) := 0$, and $g_0(d) := n+1$ for $d = 1,\ldots,c$. Then for $j = 1,\ldots,n$ the entries of g_j are computed from those of g_{j-1} by using the recursion

$$g_j(d) = \min\begin{cases} g_{j-1}(d) & \text{if } d < w_j, \\ g_{j-1}(d-w_j)+1 & \text{if } d \geq w_j. \end{cases} \tag{9.57}$$

The optimal solution value of (kSSP) is given by

$$z^* = \max_{d=0,\ldots,c}\{d \mid g_n(d) \leq k\}.$$

This means that we compute for every reachable subset value the minimum number of items summing up to this value. The recursion has time complexity $O(nc)$ and space complexity $O(kc)$, improving by a factor of k over the recursion for (kKP). Indeed, we solve the all-capacities version of problem in $O(nc)$ for all cardinalities $k' \leq k$ and all capacities $c' \leq c$.

For the exact k-item subset sum problem (E-kSSP) a special dynamic programming algorithm based on the balancing concept described in Section 3.6 was given by Pisinger [384].

Let us assume that the items are sorted according to increasing weights, and define the k-solution as the k lightest items, having the weight sum $v = \sum_{j=1}^{k} w_j$. We may assume that $v \leq c$ as otherwise no feasible solution exists. Moreover, we can identify the largest item that may be packed in a feasible solution as $m = \max\{j \mid \sum_{\ell=1}^{k-1} w_\ell + w_j \leq c\}$ and remove w.l.o.g. all items with weight larger than w_m from the input.

Using the concept of balancing one may obtain a dynamic programming algorithm with running time $O(n(c-v))$. The algorithm maintains the same cardinality k in

each iteration, hence the recursion must remove one item each time an item has been added to the knapsack.

More formally, let $g(b,\overline{w})$ for $b = k,\ldots,n$ and $v \leq \overline{w} \leq c$ be defined as

$$g(b,\overline{w}) = \max_{a=1,\ldots,b+1} \left\{ a \;\middle|\; \begin{array}{l} \text{a solution } x \text{ exists with} \\ \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{b} w_j x_j = \overline{w}, \\ (a-1) + \sum_{j=a}^{b} x_j = k, \\ x_j \in \{0,1\},\ j = a,\ldots,b \end{array} \right\} \tag{9.58}$$

where we set $g(b,\overline{w}) = 0$ if no feasible solution exists for the restricted problem.

Algorithm Balcardssp:

```
for w̄ := v+1 to c do g(k,w̄) := 0
g(k,v) := k+1
for w̄ := v+w_{k+1} to c do σ(w̄) := 1
for w̄ := c+1 to c+w_k do
    σ(w̄) := max{j | w_j < w̄ − c} + 1
for b := k+1 to n do
    for w̄ := v to c do
        g(b,w̄) := g(b−1,w̄)
    for w̄ := v to c+w_k − w_b do
        w̄' := w̄ + w_b
        if g(b−1,w̄) > σ(w̄') then
            for h := σ(w̄') to g(b−1,w̄) − 1 do
                w̄'' := w̄' − w_h
                g(b,w̄'') := max{g(b,w̄''),h}
            σ(w̄') := g(b−1,w̄)
        end if
    end for
end for
```

Fig. 9.2. Algorithm Balcardssp is a balanced dynamic programming algorithm for (E-kSSP).

At each iteration we either interchange item b with an item $h \leq k$ or we omit item b. Thus after each iteration over b all states considered will satisfy $\sum_{j=1}^{n} x_j = k$ and $\sum_{j=1}^{n} w_j x_j \leq c$. To improve the time complexity we will use a table $\sigma(\overline{w})$, for $\overline{w} = v + w_{k+1},\ldots,c + w_k$ to memorize which items have been removed previously for the corresponding weight sum $\overline{w}$, such that we do not have to repeat the same operations. This leads to the algorithm Balcardssp illustrated in Figure 9.2.

9.7.4 Approximation Algorithms

Turning from exact algorithms to approximation approaches the first point of interest concerns the LP-relaxation of (kKP) (cf. Section 2.2). Denoting as before the

optimal solution value of the LP-relaxation by z^{LP} attained for an optimal solution vector $x^{LP} := (x_1^{LP}, \ldots, x_n^{LP})$, it follows from Lemma 9.2.1 that there exists x^{LP} with at most two fractional components.

As in algorithm $H^{1/(d+1)}$ from Section 9.4.2 we let $I := \{j \mid x_j^{LP} = 1\}$ and $F := \{j \mid 0 < x_j^{LP} < 1\}$. Then the following property can be shown [65].

Lemma 9.7.2 *If $I = \{i, j\}$ and w.l.o.g. $w_j \leq w_i$, then $\sum_{r \in I} p_r + p_i \geq z^{LP}$ and the solution given by $I \cup \{j\}$ is feasible for (kKP).*

Proof. By definition of I there is $x_i^{LP} + x_j^{LP} = 1$, $p(I) + p_i x_i^{LP} + p_j x_j^{LP} = z^{LP}$ and $w_i x_i^{LP} + w_j x_j^{LP} = c - w(I)$.

Therefore, if $w_j \leq w_i$, there must be $p_i \geq p_j$, since otherwise one could improve the LP-solution by setting $x_i = 0$ and $x_j = 1$. Hence, the profits of the fractional variables can be bounded by

$$p_i x_i^{LP} + p_j x_j^{LP} \leq p_i (x_i^{LP} + x_j^{LP}) = p_i$$

showing that $z^* \leq z^{LP} \leq \sum_{r \in I} p_r + p_i$. Moreover, the weights of the fractional variables fulfill

$$w_j = w_j (x_i^{LP} + x_j^{LP}) \leq w_i x_i^{LP} + w_j x_j^{LP} = c - \sum_{r \in I} w_r,$$

which yields the feasibility of $I \cup \{j\}$. □

Lemma 9.7.2 can be immediately extended to (E-kKP). In this case the number of fractional components in the LP-solution can obviously be either 0 or 2.

Algorithm LP-Approx:

let x^{LP} be an optimal basic solution of the LP-relaxation of (kKP)
$I := \{r \mid x_r^{LP} = 1\}$
$F := \{r \mid 0 < x_r^{LP} < 1\}$ *fractional variables*
if $F = \emptyset$ then $z^A := z^{LP}$
if $F = \{i\}$ then $z^A := \max\{\sum_{r \in I} p_r, p_i\}$
if $F = \{i, j\}$ with $w_j \leq w_i$ then
$z^A := \max\{\sum_{r \in I} p_r + p_j, p_i\}$

Fig. 9.3. Approximations algorithm LP-Approx for (kKP) based on the LP-relaxation.

From Lemma 9.7.2 a straightforward approximation algorithm LP-Approx for (kKP) with solution value z^A can be constructed. It can be seen as a generalization of the classical greedy algorithm for (KP) or, to be more precise, of the simplified version Greedy-Split defined in Section 2.1. A detailed description is given

in Figure 9.3. The computation of the corresponding approximate solution vector is obvious.

Applying Lemma 9.7.2 the performance guarantee of LP-Approx follows along the same lines as Theorem 2.5.4.

Corollary 9.7.3 LP-Approx *has a tight relative performance guarantee of* $\frac{1}{2}$.

Proof. The feasibility of the computed solution is given by Lemma 9.7.2. If $F = \emptyset$, then the solution value z^A is clearly optimal. Otherwise, if $F = \{i\}$ then

$$z^* \leq z^{LP} \leq \sum_{r \in I} p_r + p_i \leq 2z^A.$$

Finally, if $F = \{i, j\}$ then

$$z^* \leq z^{LP} \leq \sum_{r \in I} p_r + p_j + p_i \leq 2z^A.$$

The tightness of the performance guarantee is established by exactly the same example as used for Ext-Greedy in Theorem 2.5.4 with $k = 3$. □

Concerning the running time, it was already mentioned in Section 9.2 that for constant d the LP-relaxation x^{LP} of (d-KP) can be computed in $O(n)$ time by an algorithm due to Megiddo and Tamir [343]. Hence, we get immediately

Corollary 9.7.4 LP-Approx *can be performed in* $O(n)$ *time.*

The adaptation of LP-Approx to (E-kKP) is straightforward. Recall that in the case of (E-kKP), F always has cardinality 0 or 2. The first case is trivial whereas in the second case the approximate solution is given by the maximum between $\sum_{r \in I} p_r + p_j$ and p_i plus the profits of the $k-1$ remaining items with smallest weight.

Note that it can be assumed w.l.o.g. that a knapsack filled with any item s and the $k-1$ remaining items with smallest weight fulfills the capacity constraint. Otherwise, item s can never be part of a feasible packing and can be removed from the set of items.

It is quite easy to construct a *PTAS* for (kKP) by applying basically the same algorithm as H^ε for (KP) from Section 2.6. Instead of procedure Ext-Greedy we plug in the above algorithm LP-Approx which has the same $\frac{1}{2}$-approximation ratio and thus permits almost the same correctness proof as for Theorem 2.6.2. Of course, the k item cardinality bound must be taken into account. The running time of such a scheme is given by $O(n^{\ell+1})$ with $\ell := \min\{\lceil \frac{1}{\varepsilon} \rceil - 2, k\}$. More details of the resulting *PTAS* and the technical modifications required for (E-kKP) can be found in [65]. Clearly, the improvements of Section 6.1.1 do not carry over to (kKP).

Recently, a *PTAS* for (kKP) with improved complexity was developed by Mastrolilli and Hutter [337]. They present an interesting technique for scaling (i.e. rounding) the item profits consisting of the combination of two basic approaches. The first one is *arithmetic rounding* which means that a profit range is partitioned into intervals of equal size and all values are rounded down to their lower interval endpoint. The second approach is *geometric rounding* where intervals have a geometrically increasing range and rounding is performed in the same way as before.

The combination of the two consists of applying arithmetic rounding to items with small profits and geometric rounding to items with large profits. The latter is based on the fact that the number of these large items which can be part of any feasible solution is naturally limited. Thus, the "error" generated by the rounding procedure can be allowed to be larger for items with large profit since only a small number of these errors will sum up in any feasible solution. Note that the scaling procedure Scaling-Reduction by Kellerer and Pferschy [267] for the *FPTAS* for (KP) described in Section 6.2.1 uses a similar argument.

Applying the combined rounding strategy Mastrolilli and Hutter [337] improve the approach by Caprara et al. [65] and attain a *PTAS* for (kKP) requiring $O(n+C(\varepsilon))$ time and $O(n)$ space. The main point is the fact that the running time is linear in n, while the constant $C(\varepsilon)$ is roughly bounded by $O((1/\varepsilon^2)^{(1/\varepsilon)})$.

Finally, we will deal with the construction of an *FPTAS* for (kKP). Caprara et al. [65] propose an algorithm relying on the dynamic programming approach given in Section 9.7.3 and follows the straightforward scaling method for (KP) in Section 2.6.

In the beginning a lower bound z^ℓ on the optimal solution value with $2z^\ell \geq z^*$ is computed, e.g. by LP-Approx. For (kKP) we define a scaled problem set with profits $\tilde{p}_j := \lfloor \frac{p_j}{K} \rfloor$ and $K := z^\ell \varepsilon \frac{1}{k}$. The optimal objective function value of the scaled instance $\tilde{z}$ can be bounded by definition of z^ℓ by

$$\tilde{z} \leq \frac{z^*}{K} = \frac{z^* k}{z^\ell \varepsilon} \leq \frac{2k}{\varepsilon}.$$

Hence, the above dynamic programming scheme with running time $O(nkU)$ can be performed in $O(nk^2/\varepsilon)$ time on the scaled instance. The space requirements are clearly given by $O(n+k^2/\varepsilon)$. The correctness of the resulting *FPTAS* can be shown without difficulties by reproducing (2.18) in a completely analogous way leading to the following statement.

Theorem 9.7.5 *There is an* FPTAS *for (kKP) with a running time in $O(nk^2/\varepsilon)$ which requires $O(n+k^2/\varepsilon)$ space.*

In principle, the more sophisticated techniques applied in Section 6.2 for (KP) could be carried over also to (kKP). However, there is a difficulty in the treatment of small items.

Recall that the separation of N into large and small items permits a more efficient scaling and reduction of the large items. Without going into details it should be a straightforward exercise to follow the basic arguments of Section 6.2.1 and partition the range of large profits between $\frac{\varepsilon}{k}U$ and U into $\frac{k}{\varepsilon}$ intervals of size $\frac{\varepsilon}{k}U$ analogous to Scaling-Reduction. In each interval only k items have to considered. Thus, the running time of the corresponding dynamic programming procedure is given by $O(k^2/\varepsilon \cdot k \cdot k/\varepsilon)$, i.e. $O(k^4/\varepsilon^2)$.

Following more closely the scaling and reduction from Section 6.2.1 and disregarding the cardinality constraint in the item reduction we can also derive a set of $O(1/\varepsilon^2)$ large items as given by Lemma 6.2.2. In this case, the dynamic programming procedure would require $O(k/\varepsilon^4)$ time. Limiting by k the number of the items in every subinterval the total number of large items can also be bounded by $O(k/\varepsilon \log(1/\varepsilon))$ yielding a total running time of $O(k^2/\varepsilon^3 \log(1/\varepsilon))$.

Note that we cannot decide which of these approaches to prefer since the ordering of running time complexities depends on the relation between n, k and $1/\varepsilon$. As discussed in Section 6.2 it is reasonable to assume n to be of larger magnitude than $1/\varepsilon$, however no such statement seems to be possible for k and $1/\varepsilon$.

In any case, after performing dynamic programming on the modified large item set, it remains to find a best possible combination of large and small items as described in Section 6.2.4. In the case of (kKP) this means that for every entry $y(q,\ell) = w$ of the dynamic programming array corresponding to a solution with profit q and consisting of ℓ items, LP-Approx must be performed to find an approximate solution with at most $k - \ell$ small items and total weight at most $c - w$. In a straightforward implementation this would require $O(n)$ time for every entry of the dynamic programming table which is either $O(k^2/\varepsilon)$ or $O(k/\varepsilon^2)$.

Unless the successive computations of LP-Approx on almost identical instances can be performed in a more efficient way, this means that the effort spent on the large items is often wasted. For the case of a large cardinality bound with $k > 1/\varepsilon$ a slightly improved algorithm with total running time $O(nk/\varepsilon^2 + k^2/\varepsilon^3 \log(1/\varepsilon))$ follows from the last-mentioned approach above. It is obvious that (E-kKP) can be tackled by almost exactly the same algorithms.

9.8 The Multidimensional Multiple-Choice Knapsack Problem

In the same way as the multiple-choice knapsack problem (MCKP) was defined as a generalization of (KP) in Section 1.2, we can generalize (d-KP) to the *multidimensional multiple-choice knapsack problem* (d-MCKP). This means that given an instance of (d-KP), the set of items N is partitioned into m mutually disjoint classes $N_1, \ldots, N_m$ and *exactly one* item of each class must be selected. A feasible selection of items is given if the d weight constraints (9.2) are fulfilled. Also the objective

function (9.1) remains unchanged. Note that unlike for (MCKP), it is nontrivial to check the existence of a feasible solution for (d-MCKP).

An interesting real-world application for (d-MCKP) was reported by van de Velde and Worm [468]. They consider a multi-period decision problem of the Dutch highway authority which wants to plan the subcontracting of highway and mainroad maintenance (about 10.000 kilometers) for a five year period. For each road a number of alternative maintenance plans are drawn up, each one requiring a different amount of budget in each of the five years. Of course, also the resulting effects on road conditions differ.

Assuming that the available budget for each of the next five years is known, the decision problem was formulated as an instance of (d-MCKP). Each class N_i corresponds to a road with the items of the class indicating the alternative maintenance strategies. One strategy has to be selected from every class (i.e. for every road), possibly including the option of doing nothing, subject to the five budget constraints and maximizing the overall effect on the quality of the road network. If the budgets are not known in advance (which is frequently the case in governmental agencies) the model can still be used to support the yearly budget negatotions.

The single-dimensional problem (MCKP) has received considerable attention as a research topic on its own right and will be covered in detail in Chapter 11. For the multidimensional case only a few publications have appeared. Due to the intrinsic difficulty of (d-MCKP), no exact algorithm has been published so far to the best of our knowledge but only heuristic approaches. Most of these are based on a heuristic for (d-KP) but employ specific elements to take the special nature of the multiple-choice constraints into account.

The first publication devoted specifically to (d-MCKP) is due to Moser, Jokanovic and Shiratori [353]. They present a heuristic algorithm based on Lagrangian multipliers. Therefore, the d weight constraints are relaxed in the same way as described in Section 9.2 with $M := \{1,\ldots,d\}$. For every set of multipliers the remaining problem (still including the multiple-choice condition) can be solved trivially by selecting the item of every class with the highest value in the new objective similar to (9.6).

The heuristic in [353] follows the main idea of Magazine and Oguz [315] as described in Section 9.5.2. Instead of packing all items, the most valuable item of each class is packed for (d-MCKP) as an infeasible initialization. As in (9.40) the most violated weight constraint i^* (after scaling) is determined. Instead of the simple removal of an item we can now only exchange an item by another item of the same class. This exchange is performed for the item which yields the smallest increase of the Lagrangian multiplier λ_{i^*} relative to the currently packed item. These exchange operations are iterated until either a feasible solution is found or no more exchanges are possible in which case the procedure terminates unsuccessfully.

As described for the dual greedy heuristics for (d-KP) it is worthwhile to add an improvement step for the determined feasible solution. This is done in [353] in

a straightforward way. For every class it is checked whether the currently packed item can be exchanged with an unpacked item of higher profit such that feasibility is preserved. After checking all classes the exchange operation with the highest increase in profits is performed (if any is found). This simple improvement step is repeated until no further exchange is possible. One may easily extend this exchange operation into removing two items of two classes instead of one. The exchange in one class yields a decrease of profits but reduces the consumption of some crucial resource. Hence, a more attractive exchange of a second class leading to an overall improvement of the objective function may be made possible. Obviously, such an improvement step is computationally more expensive.

Moser et al. [353] also present extensive computational experiments and compare their heuristic to an optimal solution derived by a (not further described) dynamic programming algorithm. This computation of an optimal solution considerably restricted the size of the tested instances. For these instances the accuracy of the approximate solutions turned out to be remarkably high.

A modified heuristic was given by Akbar et al. [8]. A parallel paper by the same working group (Khan et al. [269]) includes also an interesting application in an adaptive multimedia system. In this setup limited resources are available for a number of concurrent sessions. Each session is represented by a class where the different items of a class correspond to different levels of *quality of service* which can be assigned to a session.

In the heuristic of these authors, the least valuable items from each class are selected as a possibly infeasible initial solution. The exchange operations to move towards feasibility are based on a resource consumption measure related to the approach by Toyoda [461] as described in Section 9.5.1. Improvement steps are performed different from [353] since an exchange with any positive gain in profits is selected such that the decrease in overall resource consumption is maximal. Furthermore a double exchange step is added somehow similar to the idea sketched above.

Computational experiments by Akbar et al. [8] compare this approach with an optimal solution derived by a (not further described) branch-and-bound algorithm and with the above heuristic by Moser et al. [353]. It turns out that the modified heuristic usually delivers solutions closer to the optimum and in most cases (except for smaller instances) takes less running time than the latter.

A recent heuristic by Parra-Hernandez and Dimopoulos [371] is based on the method by Pirkul [378] (see Section 9.5.2) but includes interesting new features. The yet unpublished algorithm is fairly complicated and consistently computes better solutions than the two previously described heuristics however at the cost of considerably higher running time. In the beginning the multiple-choice constraints are relaxed to select *at most* one item of every class instead of *exactly* one. For the resulting instance of a (d-KP) with $d + m$ constraints a Lagrangian relaxation is performed and Lagrangian multipliers obtained.

With these multipliers efficiency values for the items are computed in analogy to (9.44). From the resulting ordering of the items in each class an initial solution is derived in a greedy way. If this solution is infeasible, considerable effort is spent on exchange steps to reach feasibility. Finally, improvement steps are performed where the selection criterion for the pairwise exchange of items from one class is based on a pseudo-utility value combining profit and resource consumption. For more details the upcoming paper [371] should be consulted.

Dominance relations in the obvious generalization of Section 3.4 (see also Section 11.2 for the treatment of dominance for (MCKP)) are considered in Dyer and Walker [126]. The concept of dominance allows the identification of variables which can be deleted from the problem without changing the optimal solution. As pointed out in Section 9.2 dominance and the resulting reduction is less relevant for (d-KP) than for (KP). However, the special structure of (d-MCKP) seems to allow for a more efficient application of the dominance based reduction. This is theoretically confirmed in [126] where the expected proportion of undominated variables is derived for arbitrary distributions.

10. Multiple Knapsack Problems

The *multiple knapsack problem* is a generalization of the standard knapsack problem (KP) from a single knapsack to m knapsacks with (possibly) different capacities. The objective is to assign each item to at most one of the knapsacks such that none of the capacity constraints are violated and the total profit of the items put into knapsacks is maximized.

10.1 Introduction

In the multiple knapsack problem (MKP) we are given a set of items $N := \{1,\dots,n\}$ with profits p_j and weights w_j, $j = 1,\dots,n$ and a set of knapsacks $M := \{1,\dots,m\}$ with positive capacities c_i, $i = 1,\dots,m$. We call a subset $\hat{N} \subseteq N$ *feasible* if the items of $\hat{N}$ can be assigned to the knapsacks without exceeding the capacities, i.e. if $\hat{N}$ can be partitioned into m disjoint sets N_i, such that $w(N_i) \leq c_i$ $i = 1,\dots,m$. The objective is to select a feasible subset $\hat{N}$, such that the total profit of $\hat{N}$ is maximized. Recall the formal definition of (MKP) from Section 1.2.

$$\text{(MKP)} \qquad \text{maximize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{10.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_{ij} \leq c_i\,, \quad i = 1,\dots,m, \tag{10.2}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad j = 1,\dots,n, \tag{10.3}$$

$$x_{ij} \in \{0,1\}, \quad i = 1,\dots,m, \quad j = 1,\dots,n,$$

where variable $x_{ij} = 1$ if item j is assigned to knapsack i and zero otherwise. Due to the relations with bin packing problems we will often speak of *bins* instead of knapsacks throughout this chapter. If all capacities are equal to c we speak of the *multiple knapsack problem with identical capacities* (MKP-I). The subset sum variant of (MKP) with $p_j = w_j$, $j = 1,\dots,n$, is called the *multiple subset sum problem* (MSSP) and the subset sum variant of (MKP-I) is called the *multiple subset*

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

sum problem with identical capacities (MSSP-I). We will also consider the bottleneck version of (MSSP-I), the *bottleneck multiple subset sum problem* (B-MSSP), in which the objective is to maximize the minimum load of a bin. Formally, the problem reads:

$$\begin{aligned}
\text{(B-MSSP)} \qquad \text{maximize } & \min_{i=1,\dots,m} \sum_{j=1}^{n} w_j x_{ij} \\
\text{subject to } & \sum_{j=1}^{n} w_j x_{ij} \le c, \quad i = 1,\dots,m, \\
& \sum_{i=1}^{m} x_{ij} \le 1, \quad j = 1,\dots,n, \\
& x_{ij} \in \{0,1\}, \quad i = 1,\dots,m, \quad j = 1,\dots,n.
\end{aligned}$$

Let $\tilde{M} \subseteq M$ be a subset of the knapsacks. Then $c(\tilde{M}) := \sum_{i \in \tilde{M}} c_i$ denotes the *total capacity* of the knapsacks of $\tilde{M}$. The *load* of a knapsack i will be denoted by ℓ_i and represents the overall weight of the items packed into the knapsack. Let $c_{\max} := \max\{c_1,\dots,c_m\}$ and $c_{\min} := \min\{c_1,\dots,c_m\}$ be the maximal and the minimal capacity, respectively. Additionally to the assumptions (1.16) we may assume w.l.o.g.

$$w_{\max} \le c_{\max}, \tag{10.4}$$

$$w_{\min} \le c_{\min}, \tag{10.5}$$

$$\sum_{j=1}^{n} w_j > c_{\max}. \tag{10.6}$$

Condition (10.4) guarantees that every item fits into one of the knapsacks, (10.5) ensures that for each knapsack there are items which can be put into that knapsack and finally (10.6) avoids the trivial solution that all items are assigned in one knapsack.

Let us mention that (MKP) is a special case of the so-called *generalized assignment problem* (GAP). Each item j yields profit p_{ij} (instead of profit p_j) if it is assigned to knapsack i. Thus, (GAP) can be formally defined as follows:

$$\text{(GAP)} \qquad \text{maximize } \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij}$$

$$\text{subject to } \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i = 1,\ldots,m,$$
$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad j = 1,\ldots,n, \tag{10.7}$$
$$x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \quad j = 1,\ldots,n.$$

In the literature (GAP) is often also defined with

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1,\ldots,n, \tag{10.8}$$

instead of (10.7). We prefer (10.7) since checking the feasibility of the whole item set would be already $\mathcal{NP}$-complete. A further consideration of (GAP) is omitted here, but the book by Martello and Toth [335] contains a full chapter on it.

A classical application of (MKP) can be found in cargo loading (Eilon and Christofides [129]). The problem is to choose some of n containers which have to be loaded in m vessels with different loading capacities for the shipment of the containers. Therefore, (MKP) is sometimes also called the *multiple loading problem.*

A real-world problem for (MSSP) from a company producing objects of marble was pointed out by Wirsching [483]. Every week a shipment of m marble slabs from a quarry is received by the company. These slabs have a uniform size and are much longer than wide. The company produces different products, which first have to be cut from the marble slabs and are then further processed. In particular, each product requires a piece with a specified length from a marble slab. Depending on current stock, the company prepares a list of products that it would be interested to produce. Out of this list, some products should be selected and cut from the slabs so that the total amount of wasted marble is minimized. This problem was modeled as a (MSSP) with the slabs corresponding to the knapsacks and the lengths corresponding to the weights.

This chapter does not contain a section with dynamic programming algorithms which solve (MKP) in pseudopolynomial running time. This is not surprising since in Section 10.4 it will be shown that even the simplest variants of (MKP) do not admit an *FPTAS* unless $\mathcal{P} = \mathcal{NP}$.

The rest of this chapter on (MKP) is organized as follows: Section 10.2 contains upper bounds for multiple knapsack problem. The following Section 10.3 is devoted to branch-and-bound. Approximation algorithms are the contents of Section 10.4 and a *PTAS* for the multiple knapsack problem is presented in Section 10.5. We finish in Section 10.6 with the description of two variants of (MKP).

10.2 Upper Bounds

Upper bounds for (MKP) are derived by relaxing some of the side-constraints, using either surrogate, Lagrangian or linear relaxation, as described in Section 3.8. Starting with the *surrogate relaxation*, let $\mu = (\mu_1, \ldots, \mu_m)$ be a vector of nonnegative multipliers corresponding to the m capacity constraints of (MKP). The relaxed problem $S(\text{MKP}, \mu)$ is then given by:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \\ \text{subject to} \quad & \sum_{i=1}^{m} \mu_i \sum_{j=1}^{n} w_j x_{ij} \leq \sum_{i=1}^{m} \mu_i c_i, \\ & \sum_{i=1}^{m} x_{ij} \leq 1, \qquad j = 1, \ldots, n, \\ & x_{ij} \in \{0,1\}, \qquad i = 1, \ldots, m,\ j = 1, \ldots, n. \end{aligned} \tag{10.9}$$

Martello and Toth [328] showed that the best choice of multipliers μ can be found analytically as follows:

Theorem 10.2.1 *For any instance of (MKP), the optimal choice of multipliers $\mu_1, \ldots, \mu_m$ in $S(MKP, \mu)$ is $\mu_i = k$, $i = 1, \ldots, m$, where k is a positive constant.*

Proof. Let (x_{ij}^*) be an optimal solution to $S(\text{MKP}, \mu)$ and set $t := \arg\min\{\mu_i \mid i = 1, \ldots, m\}$. A feasible solution $(\tilde{x}_{ij})$ of the same solution value can be obtained by putting all items which are assigned to a knapsack in the optimal solution, to knapsack t. Formally, we are setting $\tilde{x}_{tj} := 1$ for $x_{ij}^* = 1$ and 0, otherwise. Consequently, $S(\text{MKP}, \mu)$ is equivalent to a (KP) of the form

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_{tj} \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_{tj} \leq \left\lfloor \sum_{i=1}^{m} \mu_i c_i / \mu_t \right\rfloor, \\ & x_{tj} \in \{0,1\}, \qquad j = 1, \ldots, n. \end{aligned}$$

Since $\lfloor \sum_{i=1}^{m} \mu_i c_i / \mu_t \rfloor \geq \sum_{i=1}^{m} c_i$, setting $\mu_i := k$ for $i = 1, \ldots, m$ with k being an arbitrary positive constant, produces the minimum value of $z(S(\text{MKP}, \mu))$. □

Because of Theorem 10.2.1 the multipliers μ_i are set to $\mu_i = k$ for $i = 1, \ldots, m$ in $S(\text{MKP}, \mu)$ and the relaxed problem becomes a (KP) of the form

$$\begin{aligned} \text{maximize} & \sum_{j=1}^{n} p_j x'_j \\ \text{subject to} & \sum_{j=1}^{n} w_j x'_j \leq c, \\ & x'_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned} \tag{10.10}$$

Here $c := \sum_{i=1}^{m} c_i$, and the introduced binary variables $x'_j = \sum_{i=1}^{m} x_{ij}$ indicate whether item j has been selected in any of the knapsacks $i = 1,\ldots,m$.

A different approach is to *LP-relax* the variables x_{ij} in (MKP), leading to problem $C(\text{MKP})$. Martello and Toth [335] proved $z(C(\text{MKP})) = z(C(S(\text{MKP},1)))$, hence the objective value of the LP-relaxed problem may be found in $O(n)$ time by using algorithm Split from Section 3.1 to solve the LP-relaxation of $S(\text{MKP},1)$. The quality of the two bounds is however the same.

Two different *Lagrangian relaxations* of (MKP) are possible. By relaxing the constraints $\sum_{i=1}^{m} x_{ij} \leq 1$ using a vector $\lambda = (\lambda_1,\ldots,\lambda_n)$ of nonnegative multipliers we get the problem $L_1(\text{MKP},\lambda)$ given by:

$$\begin{aligned} \text{maximize} & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} - \sum_{j=1}^{n} \lambda_j \left(\sum_{i=1}^{m} x_{ij} - 1 \right) \\ \text{subject to} & \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \; i = 1,\ldots,m, \\ & x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \; j = 1,\ldots,n. \end{aligned} \tag{10.11}$$

By setting $\tilde{p}_j := p_j - \lambda_j$ for $j = 1,\ldots,n$ the relaxed problem can be decomposed into m independent knapsack problems, where problem i has the form

$$\begin{aligned} \text{maximize } z_i = & \sum_{j=1}^{n} \tilde{p}_j x_{ij} \\ \text{subject to} & \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \\ & x_{ij} \in \{0,1\}, \quad j = 1,\ldots,n, \end{aligned} \tag{10.12}$$

for $i = 1,\ldots,m$. All the problems have similar profits and weights, thus only the capacity distinguishes the individual instances. An optimal solution to the relaxed problem is then $z(L_1(\text{MKP},\lambda)) = \sum_{i=1}^{m} z_i + \sum_{j=1}^{n} \lambda_j$.

As opposed to surrogate relaxation, no analytical form is known for the optimal choice of multipliers λ for the Lagrangian relaxation. This means that approximate values must be found by subgradient optimization.

Assume again that the items are sorted in decreasing order of their efficiencies. Hung and Fisk [240] used predefined multipliers given by

$$\bar{\lambda}_j := \begin{cases} p_j - w_j p_s/w_s & \text{if } j < s, \\ 0 & \text{if } j \geq s \end{cases} \tag{10.13}$$

where s is the split item of $S(\text{MKP}, 1)$ and 1 is an abbreviation for the m-dimensional vector $(1,\ldots,1)$. With this choice of λ_j, in (10.12) we have $\tilde{p}_j/w_j = p_s/w_s$ for $j \leq s$, and $\tilde{p}_j/w_j \leq p_s/w_s$ for $j > s$. Hence,

$$z(C(L_1(\text{MKP},\bar{\lambda}))) = \frac{p_s}{w_s}\sum_{i=1}^{m} c_i + \sum_{j=1}^{n}\bar{\lambda}_j = \sum_{j=1}^{s-1} p_j + \left(\sum_{i=1}^{m} c_i - \sum_{j=1}^{s-1} w_j\right)\frac{p_s}{w_s}. \tag{10.14}$$

Thus we get

$$z(C(L_1(\text{MKP},\bar{\lambda}))) = z(C(S(\text{MKP}, 1))) = z(C(\text{MKP})), \tag{10.15}$$

i.e. the entries of $\bar{\lambda}$ represent the best multipliers for $C(L_1(\text{MKP},\lambda))$. In addition both $L_1(\text{MKP},\bar{\lambda})$ and the surrogate relaxation $S(\text{MKP}, 1)$ dominate the continuous relaxation. There is however no dominance between the present Lagrangian relaxation and the surrogate relaxation.

Another Lagrangian relaxation $L_2(\text{MKP},\lambda)$ appears by relaxing the weight constraints. Using a vector $\lambda = (\lambda_1,\ldots,\lambda_m)$ of nonnegative multipliers, the relaxed problem becomes

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} - \sum_{i=1}^{m}\lambda_i\left(\sum_{j=1}^{n} w_j x_{ij} - c_i\right) \\ \text{subject to } & \sum_{i=1}^{m} x_{ij} \leq 1, \\ & x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m,\ j = 1,\ldots,n. \end{aligned} \tag{10.16}$$

If one of the multipliers $\lambda_i = 0$, then the optimal solution to (10.16) will put all the items into knapsack i, which results in a useless upper bound. Hence, we may assume that $\lambda_i > 0$ for all $i = 1,\ldots,m$.

The objective function can be rewritten as

$$\text{maximize } \sum_{i=1}^{m}\sum_{j=1}^{n}(p_j - \lambda_i w_j)x_{ij} + \sum_{i=1}^{m}\lambda_i c_i, \tag{10.17}$$

which shows that the optimal solution can be found by selecting the knapsack t with the smallest associated value of λ_i and choosing all items with $p_j - \lambda_t w_j > 0$ for this knapsack. Since this is also the optimal solution of the continuous relaxation of the same problem, i.e. $z(C(L_2(\text{MKP},\lambda))) = z(L_2(\text{MKP},\lambda)))$, we get $z(L_2(\text{MKP},\lambda)) \geq z(C(\text{MKP}))$ and thus this Lagrangian relaxation cannot produce a bound tighter than the continuous one.

Note that any of the polynomially obtainable upper bounds presented in Section 5.1.1 can be used for the knapsack problems (10.10) and (10.12) to obtain upper bounds for (MKP) in polynomial time.

The most natural polynomially computable upper bound for (MKP) is

$$U_1 = \lfloor z(C(\text{MKP}))\rfloor = \lfloor z(C(S(\text{MKP},1)))\rfloor = \lfloor z(C(L_1(\text{MKP},\bar{\lambda}))\rfloor \tag{10.18}$$

Martello and Toth [335] have shown that the worst case performance of bound U_1 is $m+1$, i.e.

$$z(C(\text{MKP})) \leq (m+1)z^*, \tag{10.19}$$

and this bound is tight.

10.2.1 Variable Reduction and Tightening of Constraints

The size of a (MKP) may be reduced by preprocessing as described in Section 3.2: Assume that the solution vector corresponding to the current lower bound z^ℓ has been saved. For a given item j let u_j^0 be any upper bound on (MKP) with the additional constraint $\sum_{i=1}^m x_{ij} = 0$. If $u_j^0 \leq z^\ell$, then the constraint $\sum_{i=1}^m x_{ij} = 1$ may be added to the problem, i.e. in every improved solution to (MKP), item j must be included in some knapsack. In a similar way let u_j^1 be any upper bound on (MKP) with the additional constraint $\sum_{i=1}^m x_{ij} = 1$. If $u_j^1 \leq z^\ell$, then x_{ij} may be fixed at 0 for $i = 1,\ldots,m$. Thus in any improved solution, item j cannot be included in any of the knapsacks. Notice that the latter equation is able to fix all variables x_{ij} to their optimal value for $i = 1,\ldots,m$, while the former equation only rules out one possibility out of $m+1$. Pisinger [386] used the Dembo and Hammer bound [104] (see also (5.29))with respect to (10.10) for the reduction, getting an$O(n)$ reduction procedure. Only bounds u_j^1 were derived, attempting to exclude an item from all knapsacks. This reduction was performed at each branching node.

It may be beneficial to decrease the capacity of the individual knapsacks if it can be proved that no solution will fill a knapsack completely. Pisinger [386] presented the following technique based on the solution of a number of subset sum problems. Considering the capacity constraints (10.2) of (MKP) for each knapsack separately, the largest possible filling $\hat{c}_i$ of knapsack i is found by solving the following (SSP):

$$\hat{c}_i = \max\left\{\sum_{j=1}^n w_j x_{ij} \mid \sum_{j=1}^n w_j x_{ij} \leq c_i;\ x_{ij} \in \{0,1\},\ j = 1,\ldots,n\right\} \tag{10.20}$$

As no optimal solution x can have weight sum $\sum_{j=1}^n w_j x_{ij} > \hat{c}_i$, we may tighten the constraints in (MKP) to

$$\sum_{j=1}^{n} w_j x_{ij} \leq \hat{c}_i,\ \ i = 1,\ldots,m. \tag{10.21}$$

Since all problems (10.20) are defined for the same weights, dynamic programming may be used to derive all capacities $\hat{c}_1, \ldots \hat{c}_m$, in $O(n\bar{c})$ time where $\bar{c} = \max_{i=1,\ldots,m} c_i$.

Within a branch-and-bound algorithm it is possible to use a tighter version of the above reduction, since only variables not fixed by the enumeration need to be considered in (10.20).

10.3 Branch-and-Bound

Several branch-and-bound algorithms for (MKP) have been presented, among which we should mention Neebe and Dannenbring [357], Christofides, Mingozzi and Toth [82], Hung and Fisk [240], Martello and Toth [327], and Pisinger [386]. The first two are designed for problems with many knapsacks and few items, while the latter are best suited for problems where many items are filled into each knapsack.

Hung and Fisk [240] proposed a depth-first branch-and-bound algorithm where the upper bounds were derived by the Lagrangian relaxation, and branching was performed for the item which in the relaxed problem had been selected in most knapsacks. Each branching item was alternately assigned to the knapsacks in increasing index order, where the knapsacks were ordered in decreasing order $c_1 \geq c_2 \geq \ldots \geq c_m$. When all the knapsacks had been considered, a last branch was considered where the item was excluded from the problem.

A different branch-and-bound algorithm was proposed by Martello and Toth [327], where at each decision node, (MKP) was solved with constraint $\sum_{i=1}^{m} x_{ij} \leq 1$ omitted, and the branching item was chosen as an item which had been packed in $k > 1$ knapsacks of the relaxed problem. The branching operation generated k nodes by assigning the item to one of the corresponding $k-1$ knapsacks or excluding it from all of these. A parametric technique was used to speed up the computation of the upper bound at each node.

In a later work Martello and Toth [328] however focused on three aspects that make it difficult to solve (MKP):

- Generally it is difficult to verify feasibility of an upper bound obtained either by surrogate relaxation or Lagrangian relaxation.
- The branch-and-bound algorithm needs good lower bounds for fathoming nodes in the enumeration.
- Some knowledge to guide the branching towards good feasible solutions is needed.

In order to avoid these problems, Martello and Toth proposed a *bound-and-bound* algorithm for (MKP), named MTM. At each node of the branching tree not only an upper bound, but also a lower bound is derived. This technique is well-suited for

problems where it is easy to find a fast heuristic solution which yields good lower bounds, and where it is difficult to verify feasibility of the upper bound.

Pisinger [386] improved the framework in a number of respects: Lower bounds were derived by solving a series of subset sum problems, and subset sum problems were also used for tightening the capacity constraints of each knapsack. A well-performing (KP) algorithm for deriving upper bounds through surrogate relaxation was used, and for solving the individual subset sum problems an algorithm based on the Horowitz and Sahni decomposition is used. Moreover efficient reduction rules were applied to determine which items cannot be packed into any of the knapsacks.

10.3.1 The MTM Algorithm

Martello and Toth [328] presented the MTM bound-and-bound algorithm which is outlined in Figure 10.1.

Algorithm MTM($k, P, c_1, \ldots, c_m$):
Find an upper bound U by solving S(MKP, 1) defined on the free variables and capacity $c := \sum_{i=1}^{m} c_i$.
Find a greedy solution y giving the lower bound z'.
if $(P + z' > z^\ell)$ then
    $x^*_{ij} := x_{ij}$ for $(i,j) \in T$ and $x^*_{ij} := y_{ij}$ for $(i,j) \notin T$
    $z^\ell := P + z'$
end if
if $(P + U > z^\ell)$ then
    Choose the first item j where $y_{kj} = 1$ and item j has not been excluded from knapsack k, i.e. $(k,j) \notin T$. If no such item exists, increase k and repeat the search.
    $T := T \cup (k,j)$
    $x_{kj} := 1$ *assign item j to knapsack k*
    perform MTM($k, P + p_j, c_1, \ldots, c_k - w_j, \ldots, c_m$)
    $x_{kj} := 0$ *exclude item j from knapsack k*
    perform MTM($k, P, c_1, \ldots, c_m$)
    $T := T \setminus (k,j)$
end if

Fig. 10.1. The MTM algorithm

The algorithm derives upper bounds by solving the surrogate relaxed problem (10.10) while lower bounds are found by solving m individual knapsack problems as follows: The first knapsack $i = 1$ is filled optimally, the chosen variables are removed from the problem, and the next knapsack $i = 2$ may be filled. This process is continued until all the m knapsacks have been filled. The branching scheme follows this greedy solution, as a greedy solution should be better to guide the branching process

than individual choices at each branching node. Thus at each node two branching nodes are generated, the first one assigning the next item j of a greedy solution to the chosen knapsack i, while the other branch excludes item j from knapsack i.

Martello and Toth assume that the capacities are ordered in increasing order $c_1 \leq c_2 \leq \cdots \leq c_m$, while the items are ordered according to increasing efficiencies $e_j = p_j/w_j$. So every branching takes place on the item with the largest efficiency. At any stage, knapsacks up to k have to be filled before knapsack $k+1$ is considered. The current solution vector is x, while the optimal solution vector is denoted x^*.

At any stage the list $T = \{(i_1, j_1), \ldots, (i_d, j_d)\}$ contains indices to the variables x_{ij} which have been fixed to either 0 or 1 during the branching process. The profit sum of the currently fixed variables is P while $c_1, \ldots, c_m$ refer to the current residual capacities.

Initially variables x_{ij}, x^*_{ij} are set to 0, and the lower bound is set to $z^\ell = 0$. Then the recursion MTM$(1, 0, c_1, \ldots, c_m)$ is called. Martello and Toth used the MT1 algorithm to solve S(MKP, 1) in the first part of the algorithm.

Since every forward move setting $x_{kj} = 1$ follows the greedy solution y, the lower bound z' and solution vector y will not change by this branching. Thus z' and y need only be determined after branches of the form $x_{kj} = 0$.

10.3.2 The Mulknap Algorithm

Pisinger [386] noted that a solution to S(MKP, 1) may be validated by solving a series of subset sum problems, where the chosen items are attempted to be distributed among the m knapsacks. If this attempt succeeds, the lower bound equals the upper bound, and a backtracking occurs. Otherwise a feasible solution has been obtained which contains some (but not necessarily all) of the items selected by S(MKP, 1). Thus the algorithm Mulknap by Pisinger is merely an ordinary branch-and-bound approach since upper bounds yield a feasible solution. Pisinger also used subset sum problems for tightening the capacity constraints corresponding to each knapsack as described in Section 10.2.1. An outline of Mulknap is presented in Figure 10.2.

The branching scheme of Mulknap is based on a binary partitioning where an item j is either assigned to knapsack i or excluded from the knapsack. The knapsacks are ordered in increasing order $c_1 \leq c_2 \leq \cdots \leq c_m$ and the smallest knapsack is filled completely before starting to fill the next knapsack. At any stage, items $j \leq h$ have been fixed by the branching process, thus only items $j > h$ are considered when upper and lower bounds are determined. To keep track of which items are excluded from some knapsacks, a variable d_j for each item j indicates that the item may only be assigned to knapsacks $i \geq d_j$. The current solution vector is x while P is the profit sum of the currently fixed items.

The surrogate relaxed problem is solved using the Minknap algorithm described in Section 5.4. The subset sum problems are solved by the Decomp algorithm described in Section 4.1.4. The algorithm does not demand any special ordering of

Algorithm Mulknap$(h, P, c_1, \ldots, c_m)$:

Tighten the capacities c_i by solving m problems (SSP) defined on items $h+1, \ldots, n$.
Solve the surrogate relaxed problem $S(\text{MKP}, 1)$ with $c := \sum_{i=1}^{m} c_i$.
Let $\hat{y}$ be the corresponding solution and U its objective value.
if $(P + U > z^\ell)$ then
    Split the solution $\hat{y}$ in the m knapsacks by solving a series of (SSP) defined on items with $\hat{y}_j = 1$.
    Let (y_{ij}) be the optimal filling of knapsack i with corresponding profit sum z_i.
    Improve the heuristic solution by greedy filling knapsacks with $\sum_{j=h+1}^{n} w_j y_{ij} < c_i$.
    if $(P + \sum_{i=1}^{m} z_i > z^\ell)$ then
        $x^*_{ij} := y_{ij}$ for $j \geq h$ and $x^*_{ij} := x_{ij}$ for $j < h$
        $z^\ell := P + \sum_{i=1}^{m} z_i$
    end if
end if
if $(P + U > z^\ell)$ then
    Reduce the items as described in Section 10.2.1, and swap the reduced items to the first positions, increasing h.
    Let i be the smallest knapsack with $c_i > 0$.
    Solve a (KP) defined on the free variables with $c := c_i$.
    Denote the solution vector by $\bar{y}$.
    Choose the branching item k as the item with largest efficiency among items with $\bar{y}_j = 1$.
    Swap item k to position $h+1$ and set $j := h+1$.
    $x_{ij} := 1$ *assign item j to knapsack i*
    Perform Mulknap$(h+1, P + p_j, w_j, c_1, \ldots, c_i - w_j, \ldots, c_m)$.
    $x_{ij} := 0$, $d' := d_j$, $d_j := i+1$ *exclude j from knapsack i*
    Perform Mulknap$(h, P, c_1, \ldots, c_m)$.
    Find j again, and set $d_j := d'$.
end if

Fig. 10.2. The Mulknap algorithm.

the variables, as the Minknap algorithm makes the necessary ordering itself. This however implies that items are permuted at each call to Minknap. Thus the last line of Mulknap cannot assume that item j is at the same position as before.

The main algorithm thus only has to order the capacities according to increasing c_i, set $z^\ell = 0$ and initialize $d_j = 1$, and $x^*_{ij} = y_{ij} = 0$ for all i, j before calling Mulknap$(0, 0, c_1, \ldots, c_m)$.

10.3.3 Computational Results

We will compare the performance of the MTM algorithm with that of the Mulknap algorithm. Four different types of randomly generated instances are considered for different ranges $R = 100,\ 1000$ and 10000.

- *uncorrelated instances*: p_j and w_j are randomly distributed in $[1,R]$.
- *weakly correlated instances*: w_j randomly distributed in $[1,R]$ and p_j randomly distributed in $[w_j - R/10, w_j + R/10]$ such that $p_j \geq 1$.
- *strongly correlated instances*: w_j randomly distributed in $[1,R]$ and $p_j = w_j + 10$.
- *subset sum instances*: w_j randomly distributed in $[1,R]$ and $p_j = w_j$.

Martello and Toth [335] proposed to consider two different classes of capacities as follows: *Similar capacities* have the first $m-1$ capacities c_i randomly distributed in

$$\left[\frac{4}{10}\sum_{j=1}^{n}\frac{w_j}{m},\ \frac{6}{10}\sum_{j=1}^{n}\frac{w_j}{m}\right] \quad \text{for } i = 1,\ldots,m-1, \tag{10.22}$$

while *dissimilar capacities* have c_i distributed in

$$\left[0,\ \frac{1}{2}\left(\sum_{j=1}^{n} w_j - \sum_{k=1}^{i-1} c_k\right)\right] \quad \text{for } i = 1,\ldots,m-1. \tag{10.23}$$

The last capacity c_m is in both classes chosen as

$$c_m := \frac{1}{2}\sum_{j=1}^{n} w_j - \sum_{i=1}^{m-1} c_i, \tag{10.24}$$

to ensure that the sum of the capacities is half of the total weight sum. For each instance a check is made on whether the assumptions (10.4) to (10.6) are violated in which case a new instance is generated.

The codes were run on a INTEL PENTIUM III, 933 MHZ with a maximum amount of 1 hour assigned to each algorithm for solving the 20 instances in each class. A dash in the following tables indicates that the 20 problems could not be solved within this time limit. The MTM algorithm, which originally is designed for problems up to $n = 1000$, was extended such that problems up to $n = 100000$ could be solved. Also the Mulknap algorithm — demanding less static space — was run with instances up to $n = 100000$. All the instances were tested with $m = 10$ knapsacks. In each of the following tables, instances with similar capacities are considered in the upper part of the table, while the lower part of the table considers instances of dissimilar capacities. For computational experiments with instances for small values of n we refer to Pisinger [386].

	Uncorrelated			Weakly correlated			Strongly correlated			Subset sum		
$n \setminus R$	100	1000	10 000	100	1000	10 000	100	1000	10 000	100	1000	10 000
100	0.24	40.89	—	5.72	—	—	—	—	—	0.00	0.00	0.01
300	0.02	3.52	55.04	0.01	30.20	—	—	—	—	0.00	0.00	0.01
1000	0.00	5.25	74.74	0.00	7.63	—	—	—	—	0.00	0.00	0.01
3000	0.01	1.51	—	0.01	1.24	—	—	—	—	0.01	0.01	0.02
10000	0.15	1.46	—	0.07	0.31	—	—	—	—	0.07	0.07	0.07
30000	4.30	0.21	—	0.21	0.22	2.86	—	—	—	0.22	0.22	0.23
100000	80.48	18.70	—	—	0.86	1.41	—	—	—	0.85	0.86	0.87
100	0.02	0.12	1.94	0.07	4.15	53.04	33.98	111.93	—	0.00	0.12	27.28
300	0.01	0.23	2.88	0.01	8.99	110.89	—	—	—	0.00	0.02	0.29
1000	0.01	0.23	6.20	0.00	1.58	—	—	—	—	0.00	0.00	0.23
3000	0.08	0.37	21.19	0.02	0.44	—	—	—	—	0.01	0.01	0.04
10000	0.63	0.41	36.26	0.13	0.26	—	—	—	—	0.08	0.08	0.08
30000	—	0.39	53.51	19.24	0.25	18.69	—	—	—	0.25	0.24	0.25
100000	—	1.41	26.53	—	0.97	1.29	—	—	—	0.98	0.95	0.89
	Top: similar capacities. Bottom: dissimilar capacities.											

Table 10.1. Computing time in seconds, MTM, as average of 20 instances (INTEL PENTIUM III, 933 MHz).

	Uncorrelated			Weakly correlated			Strongly correlated			Subset sum		
$n \setminus R$	100	1000	10 000	100	1000	10 000	100	1000	10 000	100	1000	10 000
100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.01
300	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.23	0.00	0.00	0.01
1000	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.07	1.23	0.00	0.00	0.01
3000	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.27	5.56	0.00	0.00	0.01
10000	0.00	0.00	0.00	0.00	0.00	0.01	0.09	1.19	42.76	0.00	0.00	0.01
30000	0.01	0.01	0.01	0.01	0.01	0.01	0.34	7.09	123.77	0.00	0.01	0.01
100000	0.04	0.04	0.05	0.04	0.03	0.04	1.46	30.69	302.04	0.03	0.03	0.04
100	0.00	0.00	0.07	0.00	0.05	2.29	0.00	0.22	2.18	0.00	0.03	2.16
300	0.00	0.00	0.01	0.00	0.00	0.02	0.00	0.35	0.82	0.00	0.02	0.07
1000	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.05	14.57	0.00	0.00	0.03
3000	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.32	8.63	0.00	0.00	0.01
10000	0.00	0.00	0.00	0.00	0.00	0.01	0.07	1.26	43.01	0.00	0.00	0.01
30000	0.01	0.01	0.01	0.01	0.01	0.01	0.32	6.56	105.15	0.01	0.01	0.01
100000	0.04	0.04	0.05	0.05	0.05	0.06	1.63	30.12	299.02	0.05	0.05	0.06
	Top: similar capacities. Bottom: dissimilar capacities.											

Table 10.2. Computing time in seconds, Mulknap, as average of 20 instances (INTEL PENTIUM III, 933 MHz).

The two tables, Table 10.1 and 10.2 compare the solution times of MTM with those of Mulknap. The running times indicate, that small instances with $n = 100$ are difficult, while larger instances become easier to solve, since there many items in each knapsack and hence it is easiler to obtain solutions which use all of the capacity c_i. Very-large sized instances, again tend to be more difficult. For MTM there are a couple of large-sized instances which cannot be solved, while Mulknap has a more controlled growth of the solution times.

Naturally, MTM cannot solve strongly correlated knapsack problems of larger size, since it is based on simple LP-based upper bounds, which perform badly on these instances. Using better upper bounds from Section 5.1.1 could improve on this.

For very large instances $n \geq 1000$, Mulknap is actually able to solve the problems in times comparable to the best solution times for ordinary (KP).

For computational experiments with the two codes on instances with only a few items, we refer to the paper by Pisinger [386].

10.4 Approximation Algorithms

It is well-known that even the special multiple knapsack problems (MSSP-I) and (B-MSSP) are strongly $\mathcal{NP}$-hard as they are both optimization versions of the strongly $\mathcal{NP}$-complete *3-partitioning problem* [164, SP15] denoted as 3-PART. (For a definition of 3-PART we refer to the proof of Theorem 10.4.5.) This rules out the existence of pseudopolynomial algorithms or fully polynomial time approximation schemes for the variations of (MKP) presented in Section 10.1 unless $\mathcal{P} = \mathcal{NP}$ holds. By using again a reduction from EQUIPARTITION as in Theorem 9.4.1 for the 2-dimensional knapsack problem, it can be easily shown that this holds even for $m = 2$.

Theorem 10.4.1 *If the number of knapsacks m is equal to 2, neither (MSSP-I) nor (B-MSSP) admit an* FPTAS *unless* $\mathcal{P} = \mathcal{NP}$.

Proof. We have given an instance of EQUIPARTITION with n items of integer weight a_j for $j = 1,\ldots,n$, n even and $\sum_{j=1}^{n} a_j = 2A$. (for the exact definition see the proof of Theorem 9.4.1). Then define an instance of (MSSP-I) with two knapsacks, n items, the j-th having weight $w_j = 2A + a_j$, $j = 1,\ldots,n$, and knapsack capacity $c = A(n+1)$.

Clearly, EQUIPARTITION has a solution if and only if the optimal solutions of (MSSP-I) and (B-MSSP) pack all items in the knapsacks, namely $n/2$ per knapsack, and the overall load of the knapsacks is $2A(n+1)$. Observe that $n/2$ is the maximum number of items that can be packed in a knapsack. Any solution of (MSSP-I) or (B-MSSP) that is not optimal, leaves at least one item unpacked. Hence, the overall load of the knapsacks is at most $2An$, and the smallest load of a knapsack at most An. Thus, the approximation ratio of any nonoptimal solution is at most

$$\frac{2An}{2A(n+1)} = 1 - \frac{1}{n+1}$$

for (MSSP-I) and

$$\frac{An}{A(n+1)} = 1 - \frac{1}{n+1}$$

for (B-MSSP). If an *FPTAS* existed for either problem, we would get a polynomial $(1-\varepsilon)$-approximation algorithm also for $\varepsilon < \frac{1}{n+1}$, which would yield an optimal solution for the instance above whenever the original EQUIPARTITION instance has a solution. This is clearly impossible unless $\mathcal{P} = \mathcal{NP}$. □

Note that for (B-MSSP) an even stronger result is presented in Thereom 10.4.5.

10.4.1 Greedy-Type Algorithms and Further Approximation Algorithms

A straightforward approach to obtain an approximation algorithm for (MKP) is generalizing the greedy approach of Section 2.1 by taking the linear programming relaxation of Section 2.2 to produce a feasible solution. Assume the items to be sorted by their efficiency in decreasing order. Then the linear programming relaxation of (MKP) assigns the items to the knapsacks in this order, possibly "breaking" at most m split items. This linear programming relaxation is then transformed into a feasible solution z^G for (MKP) by removing all these split items. For each knapsack i define the *split item* s_i as

$$s_i := \min\left\{k \mid \sum_{j=1}^{k} w_j > \sum_{\ell=1}^{i} c_\ell\right\}. \tag{10.25}$$

Then knapsack i is filled with items $s_{i-1}+1, s_{i-1}+2, \ldots, s_i - 1$ setting $s_0 := 0$. The absolute error of z^G becomes less than $\sum_{i=1}^{m} p_{s_i}$. By the same arguments as for (KP) the relative error of this greedy heuristic can be arbitrarily bad.

It is not difficult to adapt the greedy approach in order to get an algorithm with relative performance guarantee 1/2. We call this algorithm the *generalized greedy algorithm* G-Greedy. Assume the capacities of the knapsacks to be sorted in increasing order, i.e.

$$c_1 \leq c_2 \leq \ldots \leq c_m. \tag{10.26}$$

Recall that the items are sorted by their efficiencies in decreasing order. Now assign items in this order to the knapsacks in the order $1, \ldots, m$. Allow an item j only to be "assigned" to knapsacks i where it fits in, i.e. to knapsacks i with $w_j \leq c_i$. This is realized by removing items from the list as long as their weight is greater than the capacity of the current knapsack. By enlarging the linear programming relaxation of (MKP) with the additional constraints $x_{ij} = 0$ for all items j and knapsacks i with $w_j > c_i$, we can be sure that the i-th split item s_i fits into knapsack i, $i = 1, \ldots, m$, and the m split items $s_1, \ldots, s_m$ form a feasible solution. Algorithm G-Greedy now takes the solution which forms the maximum of the greedy solution and the total profit of the split items.

Theorem 10.4.2 *For (MKP) algorithm* G-Greedy *has a tight relative performance ratio of 1/2.*

Proof. Since the absolute error between optimal solution and greedy solution is bounded from above by the total profit of the split items, we obtain

$$z^* - z^{GG} \leq \sum_{i=1}^{m} p_{s_i} \leq z^*,$$

which yields the desired performance guarantee. Several instances for (MSSP-I) can be taken to show the tightness. For example take m items of weight $c/2+\varepsilon$ used by the heuristic solution and m items of weight c used by the optimal solution. □

Another greedy strategy $\mathsf{G}(\varepsilon)$ was proposed by Chekuri and Khanna [79]. Their algorithm $\mathsf{G}(\varepsilon)$ packs knapsacks one at a time, by applying an *FPTAS* for the single knapsack problem on the remaining items. The parameter ε refers to the error tolerance used in the *FPTAS* for (KP).

Theorem 10.4.3 *For (MKP) algorithm* $\mathsf{G}(\varepsilon)$ *has a tight relative performance guarantee of* $1/2-\varepsilon/2$.

Proof. Let Z_i^* denote the items that are assigned to bin i in some optimal solution but which are not used by $\mathsf{G}(\varepsilon)$. Let Z_i denote the set of items put into bin i by $\mathsf{G}(\varepsilon)$. Then, $p(Z_i) \geq (1-\varepsilon)p(Z_i^*)$, since all items of Z_i^* are available for $\mathsf{G}(\varepsilon)$. Adding the profits we get

$$\sum_{i=1}^{m} p(Z_i) \geq (1-\varepsilon)\sum_{i=1}^{m} p(Z_i^*).$$

If $\sum_{i=1}^{m} p(Z_i^*) \geq z^*/2$, we are done. For $\sum_{i=1}^{m} p(Z_i^*) < z^*/2$ algorithm $\mathsf{G}(\varepsilon)$ must have packed the remaining items of the optimal solution which again have total profit at least $z^*/2$.

The tightness example is given by the following instance of (MSSP). We are given two bins with capacities $c_1 > c_2$, two items with weights c_2 and c_1 and identical profits equal to 1. If $\mathsf{G}(\varepsilon)$ packs the item with weight c_2 into the first knapsack, we have $z^* = 2$ but $z^{G(\varepsilon)} = 1$. □

For problem (MKP-I) Chekuri and Khanna [79] have shown that $\mathsf{G}(\varepsilon)$ has a better worst-case behavior. They could show:

Theorem 10.4.4 *For (MKP-I) algorithm* $\mathsf{G}(\varepsilon)$ *has a relative performance guarantee of* $(1-\frac{1}{e}-O(\varepsilon)) \approx 0.632 - O(\varepsilon)$.

Proof. Let Z^* denote the set of items that are assigned to the bins in some optimal solution and let Z_i^* denote those items in Z^* which remain unpacked by $\mathsf{G}(\varepsilon)$ after the algorithm has packed the first $(i-1)$ bins. The set Z_i shall contain the items assigned to the i-th bin by $\mathsf{G}(\varepsilon)$. Let

$$G_i := \sum_{\ell=1}^{i} p(Z_\ell)$$

denote the profit of $G(\varepsilon)$ after packing the first i bins. Since the bin capacities are equal, we get by an averaging argument and from $z^* \leq p(Z_i^*) + G_{i-1}$ that

$$p(Z_i) \geq (1-\varepsilon)p(Z_i^*)/m \geq (1-\varepsilon)(z^* - G_{i-1})/m.$$

This result can be used to show inductively that

$$G_i \geq \left(1 - \left(1 - \frac{1-\varepsilon}{m}\right)^i\right) z^*.$$

Hence it follows that

$$G_m \geq \left(1 - \left(1 - \frac{1-\varepsilon}{m}\right)^m\right) z^* \geq (1 - e^{\varepsilon-1}) z^*.$$

Expanding e^{ε} and ignoring the higher order terms we get the desired result. □

For (MSSP-I) a 1/2-approximation algorithm can be obtained much easier by the classical First-Fit-Decreasing (FFD) strategy. It packs the items in decreasing order of weight into the first knapsack where they fit, yielding a $1/2$-approximation algorithm, since either every bin is filled by at least $c/2$ or all items are packed.

In Section 10.4.2 a $2/3$-approximation algorithm for (B-MSSP) will be described, which can be easily adapted to the sum version and runs in $O(n \log n)$ time. A stronger improvement of the worst-case performance guarantee without considerable increase of the running time poses a non-trivial problem. In [64] Caprara, Kellerer and Pferschy present a $3/4$-approximation algorithm with a very competitive running time of $O(m^2 + n)$.

Another approximation algorithm for (MKP) was investigated by Martello and Toth [329]. The items are assumed to be sorted by their efficiency in decreasing order and the capacities in increasing order according to (10.26). Analogously to $G(\varepsilon)$ their algorithm packs knapsacks one at a time, but here in increasing order of the capacities. Instead of an *FPTAS* the algorithm Ext-Greedy for single knapsack problems introduced in Section 2.5 is applied on the remaining items. After the greedy steps the algorithm looks for improvements by a local exchange strategy. If two items assigned to different knapsacks can be interchanged such that one more item fits into the knapsack, then this exchange is made. The resulting algorithm runs in $O(n^2)$ time. In [335] computational results are presented for randomly generated instances, but no worst-case analysis is given.

10.4.2 Approximability Results for (B-MSSP)

Whereas for (MKP) one can find a solution which is arbitrarily close to the optimum in polynomial time, as will be shown in Section 10.5, for the case of (B-MSSP) only a 2/3-approximation algorithm can be given. This is also the best possible performance ratio achievable in polynomial time unless $\mathcal{P} = \mathcal{NP}$. Both results are contained in a paper by Caprara, Kellerer and Pferschy [62].

Theorem 10.4.5 *There does not exist any polynomial time* $(2/3+\delta)$*-approximation algorithm for (B-MSSP) for any* $\delta > 0$ *unless* $\mathcal{P} = \mathcal{NP}$.

Proof. Consider the well-known $\mathcal{NP}$-complete *3-partitioning* [164] problem, which is defined as follows:

3-partitioning (3-PART)
Instance: $3m+1$ numbers $a_1,\ldots,a_{3m},b$ such that $\sum_{j=1}^{3m} a_j = mb$ and $b/4 < a_j \leq b/2$ for $j = 1,\ldots,3m$.
Question: Is there a partition of $\{1,\ldots,3m\}$ into sets $S_1,\ldots,S_m$ such that $|S_i| = 3$ and $\sum_{j\in S_i} a_j = b$ for $i = 1,\ldots,m$?

Given an instance of 3-PART, we can define a (B-MSSP) instance with $3m$ items and m bins, with item weights $a_j+\gamma$, $j = 1,\ldots,3m$, and knapsack capacity $3\gamma+b$. If $\gamma > 3a_{\max}$, with $a_{\max} := \max\{a_j \mid j = 1,\ldots,3m\}$, at most three items can be packed into each knapsack. Then, if the original 3-PART instance has a solution, the optimal value of the (B-MSSP) instance constructed is $3\gamma+b$, otherwise its optimal value is at most $2\gamma+2a_{\max} \leq 2\gamma+b$. Hence, assuming the existence of a polynomial time $(2/3+\delta)$-approximation algorithm for (B-MSSP), by taking γ sufficiently large we get a polynomial time algorithm for 3-PART. □

A simple 2/3-approximation algorithm for (B-MSSP), called $H^{\frac{2}{3}}$ was presented in [62]. It is illustrated in Figure 10.3. Algorithm $H^{\frac{2}{3}}$ basically packs items with weight larger than $2/3\,c$ into separate bins, then packs a maximal number of bins with pairs consisting of items with weight between $1/3\,c$ and $2/3\,c$. Finally the remaining items are considered in decreasing order of weight and each item is packed into the bin with smallest weight. This is the well-known rule of *longest processing time* (LPT) used in scheduling problems with identical parallel machines. The algorithm stops as soon as each bin has a load not smaller than $2/3\,c$.

Formally, the item set N is partitioned into the set $N_1 := \{j \mid w_j \geq 2/3\,c\}$ of *large* items, the set $N_2 := \{j \mid c/3 < w_j < 2/3\,c\}$ of *medium* items, and the set $N_3 := \{j \mid w_j \leq c/3\}$ of *small* items. Let $n_1 := |N_1|$, $n_2 := |N_2|$ and $n_3 := |N_3|$. Note that the computation of the maximum number of pairs for a given item set T can be performed in $O(|T|\log|T|)$ time by standard techniques, and will not be described in detail.

The following result was proved by Csirik, Kellerer and Woeginger in [95] in the context of assigning a set of jobs to m identical processors in order to maximize the earliest processor completion time. We restate it in terms of (B-MSSP).

Lemma 10.4.6 *Suppose* $c = \infty$. *Then, by assigning the items in decreasing order of weights, each to the bin with smallest load, the (B-MSSP) solution value obtained (i.e. the minimum load of the knapsacks) is at least* $(3m-1)/(4m-2)$ *times the optimal one.* □

Algorithm $H^{\frac{2}{3}}$:

Initialization:
Partition the item set N into N_1, N_2 and N_3 computing n_1, n_2, n_3.
Pack each item in N_1 into an empty bin, $m_1 := n_1$.

Phase 1: *Pack the medium items*
If $n_2 \leq m - m_1$ then
    pack each item in N_2 into an empty bin and goto Phase 2.
$max_2 := n_2 - m - m_1$ *upper bound on the number of pairs*
Let T contain the at most max_2 smallest items in N_2.
Compute the maximum number m_2 of pairs obtainable from items in T and pack each pair into an empty bin.
Pack each of the $m - m_1 - m_2$ largest items in N_2 into an empty bin.

Phase 2: *Pack the small items*
while $N_3 \neq \emptyset$ do
    let j be the largest item in N_3
    let b be the bin with smallest load ℓ_b
    if $\ell_b \geq 2/3\, c$ then stop
    pack item j in bin b
end while

Fig. 10.3. A 2/3-approximation algorithm for (B-MSSP).

Using this result Caprara, Kellerer and Pferschy [62] prove that the approximation guarantee of Algorithm $H^{\frac{2}{3}}$ is best possible unless $\mathcal{P} = \mathcal{NP}$.

Theorem 10.4.7 *Algorithm $H^{\frac{2}{3}}$ has a relative performance guarantee of 2/3 for (B-MSSP). It runs in $O(n \log n)$ time and requires $O(n)$ space.*

Proof. If load $\ell_i \geq 2/3\, c$ for each bin i in the $H^{\frac{2}{3}}$ solution, the claim is clearly true. We will assume that this is not the case, which implies in particular that all the items in N_3 are packed by $H^{\frac{2}{3}}$. We will also assume $N_1 = \emptyset$. Indeed, if $N_1 \neq \emptyset$, then for the instance defined by item set $N \setminus N_1$ and with $m - n_1$ bins the optimal solution is not worse than z^* and the value of the solution returned by $H^{\frac{2}{3}}$ is unchanged.

Let z^H denote the value of the $H^{\frac{2}{3}}$ solution, R be the set of bins in this solution containing at most one item in N_2, and $r := |R|$. By the assumption above and the structure of Phase 2, all the items in N_3 are packed in some bin in R. Let q be the number of items in N_2 packed in the bins in R, and denote by $x_1, \ldots, x_q$ the weights of these items, in decreasing order.

We now claim that there exists a set R^* of r bins in the optimal solution where $p \leq q$ items in N_2 are packed, and such that the weights of these items, say $y_1, \ldots, y_p$, in decreasing order, satisfy $y_j \leq x_j$ for $j = 1, \ldots, p$.

If the number of pairs of items in N_2 packed by the optimal solution is not larger than m_2, the number of pairs packed by $H^{\frac{2}{3}}$, then the claim follows immediately as the largest items in N_2 are packed alone in a bin by $H^{\frac{2}{3}}$.

Otherwise, if the optimal solution contains more pairs of items in N_2, say $m_2^* > m_2$, then it also contains at least $m_2^* - m_2$ more bins with no item in N_2 packed, as the number of pairs in $H^{\frac{2}{3}}$ is the maximum that guarantees that as many bins as possible contain at least one item in N_2.

Now, consider the (B-MSSP) instance in which the capacity is equal to ∞, the number of bins is r, and the item set is given by the items packed by $H^{\frac{2}{3}}$ in the bins in R, whose weights, in decreasing order, we denote by $x_1, \ldots, x_t$. Let $\tilde{z}$ denote the optimal solution value for this instance. Clearly, $\tilde{z} \geq z^*$, as the items packed by the bins in R^* by the optimal solution, whose weights are (in decreasing order) $y_1, \ldots, y_s$, satisfy $s \leq t$ and $y_j \leq x_j$ for $j = 1, \ldots, s$. Moreover, the packing of the bins in R by $H^{\frac{2}{3}}$ yields a solution $z^H \geq (3r-1)/(4r-2)\,\tilde{z} \geq 3/4\,z^*$ by Lemma 10.4.6, which concludes the proof. □

10.5 Polynomial Time Approximation Schemes

Until recently, not much was known about *PTAS* for multiple knapsack problems. In 1999 Caprara, Kellerer and Pferschy derived a *PTAS* for (MSSP-I) [62] which was generalized to (MSSP) a short time later by the same authors [63]. Then, Kellerer presented a *PTAS* for (MKP-I) [263]. Finally, these results have been generalized by Chekuri and Khanna [79] who gave a *PTAS* for (MKP). We will outline the proofs for (MSSP) and (MKP). While the proof of (MSSP) can be explained easily in the context of Chapter 4, the techniques used by Chekuri and Khanna are more sophisticated and a detailed description would be beyond of the scope of this book.

10.5.1 A *PTAS* for the Multiple Subset Problem

The *PTAS* for (MSSP) presented in [63] is developed in several levels. First, a *PTAS* is given for a special case of (MSSP), namely the *capacity dense* (MSSP), where the ratio between the maximum and minimum capacity of a knapsack is bounded by a constant. Using this algorithm a polynomial time $(1-\varepsilon)$-approximation algorithm is constructed for the more general capacity grouped (MSSP) which is a (MSSP) where the knapsack capacities can be partitioned such that a bounded number of capacity dense subproblems arises. Finally, a *PTAS* for the general (MSSP) is derived by artificially generating a number of capacity grouped instances and taking the best solution obtained by applying the $(1-\varepsilon)$-approximation to each of these instances.

We start with the *PTAS* for the *capacity dense* (MSSP). This first *PTAS* will be referred to by $PTAS_{CD}$.

Let ℓ be the smallest integer such that $c_{\min} \geq \tilde{\varepsilon}^{\ell} c_{\max}$ for some small constant $\tilde{\varepsilon} > 0$. Note that the capacity dense requirement ensures that ℓ is bounded by a constant. As usual partition the item set N into the set $L := \{j \in N \mid w_j > \tilde{\varepsilon}^{\ell+1} c_{\max}\}$ of *large* items and $S := \{j \in N \mid w_j \leq \tilde{\varepsilon}^{\ell+1} c_{\max}\}$ of *small* items.

As in Lemma 4.5.1 it can be easily shown that every polynomial time $(1-\tilde{\varepsilon})$-approximation algorithm for instances with large items can be extended to a polynomial time $(1-\tilde{\varepsilon})$-approximation algorithm for general instances by adding the small items by a simple greedy procedure. This allows one to initially get rid of the small items, which are reconsidered only at the end of the algorithm. At the same time, one may consider only the large items when the performance of an algorithm is analyzed.

After having (temporarily) removed the small items, the set of large items L is reduced to the so-called set of *relevant items* R and the other items are discarded. This is done in the following way: L is partitioned into subsets

$$I_j :=]j\,\tilde{\varepsilon}^{\ell+1} c_{\max}, (j+1)\tilde{\varepsilon}^{\ell+1} c_{\max}\,], \quad j = 1, \ldots, \left\lceil \frac{1}{\tilde{\varepsilon}^{\ell+1}} \right\rceil - 1. \tag{10.27}$$

Let

$$\sigma_j := \left\lceil \frac{1}{j\tilde{\varepsilon}^{\ell+1}} \right\rceil - 1. \tag{10.28}$$

If $|I_j| \leq 2m\sigma_j$, we select all items in I_j, otherwise we select only the $m\sigma_j$ largest and the $m\sigma_j$ smallest and put them into R. Using $\sigma_1 = \left\lceil \frac{1}{\tilde{\varepsilon}^{\ell+1}} \right\rceil - 1$ we conclude

$$\begin{aligned} r := |R| &\leq \sum_{j=1}^{\sigma_1} 2m\sigma_j \\ &= \sum_{j=1}^{\sigma_1} 2m \left(\left\lceil \frac{1}{j\tilde{\varepsilon}^{\ell+1}} \right\rceil - 1 \right) \\ &\leq 2m \frac{1}{\tilde{\varepsilon}^{\ell+1}} \left(\ln \left(\frac{1}{\tilde{\varepsilon}^{\ell+1}} \right) + 1 \right), \end{aligned} \tag{10.29}$$

and hence r is in $O(m)$.

The definition of R is analogous to the definition of relevant items for (SSP) in Section 4.6. Thus, it can be shown analogously to Lemma 4.6.1 that any optimal solution z_R^* for the relevant items R is at most εc smaller than an optimal solution z_L^* for the large items, i.e.

$$z_R^* \geq (1-\varepsilon) z_L^*. \tag{10.30}$$

Hence, we only have to find a near-optimal solution for the instance defined by R. In [63] it is distinguish between two cases, depending on the relative values of m

and $\tilde{\varepsilon}$. If $m \leq 3/\tilde{\varepsilon}^{\ell+1}$, then by (10.29) the number r of items as well as the number of knapsacks are bounded by a constant, and an optimal solution can be found in constant time by complete enumeration.

In the remainder, the more difficult case $m > 3/\tilde{\varepsilon}^{\ell+1}$ is considered. Again a new instance, with only a *constant* number of different weights, is introduced. The basic idea for this is the *shifting technique*, an idea which was introduced by Fernandez de la Vega and Luecker [142] for bin packing. Define

$$k := \left\lfloor m\tilde{\varepsilon}^{\ell+1} \right\rfloor \tag{10.31}$$

and let p and q be such that $r = pk + q$ and $1 \leq q \leq k$. Note that $k \geq 3$. It is easy to check that

$$p \leq \frac{3}{\tilde{\varepsilon}^{2\ell+2}} \left(\ln\left(\frac{1}{\tilde{\varepsilon}^{\ell+1}}\right) + 1 \right) \tag{10.32}$$

and hence p is bounded by a constant.

Denote by $1,\ldots,r$ the items in R and assume that they are sorted in increasing order of weights, i.e. $w_1 \leq w_2 \leq \ldots \leq w_r$. Partition R into the $p+1$ subsets $R_j := \{jk+1,\ldots,(j+1)k\}$, $j = 0,\ldots,p-1$, containing k items each and $R_p := \{pk+1,\ldots,r\}$ containing $q \leq k$ items.

Define the (MSSP) instance I with k items of weight $v_j := w_{(j+1)k}$, $j = 0,\ldots,p-1$, and q items of weight $v_p := w_r$. Because the number of distinct item weights p is constant, this instance can be solved to optimality in polynomial time as follows.

We call a *feasible knapsack filling* a vector t with $p+1$ nonnegative integer entries $t_0,\ldots,t_p$, such that $t_j \leq k$ for $j = 0,\ldots,p-1$, $t_p \leq q$, and

$$\sum_{j=0}^{p} t_j v_j \leq c_{\max}. \tag{10.33}$$

Let f denote the number of different knapsack fillings. By a rough estimate f can be bounded by $\left\lfloor \frac{1}{\tilde{\varepsilon}^{\ell+1}} \right\rfloor^{(p+1)}$ and hence by a constant, because in any knapsack there can be at most $\left\lfloor \frac{1}{\tilde{\varepsilon}^{\ell+1}} \right\rfloor$ items with weight v_i, $i = 0,\ldots,p$.

Since every knapsack filling can be assigned to at most m knapsacks, we have at most m^f feasible assignments with different weights of items of I to the m knapsacks. As the number f of different knapsack fillings is constant, the optimal solution for instance I can be found in polynomial time by complete enumeration.

The solution obtained for instance I is converted into a solution for the item set R (and hence item set L) by replacing the s_j items of weight v_j in the solution, $j = 0,\ldots,p$, by items $(j+1)k,(j+1)k-1,\ldots,(j+1)k-s_j+1$. Denoting the value of this solution by z_L^H, the next lemma evaluates its quality.

Lemma 10.5.1 $z_L^H \geq (1-4\tilde{\varepsilon})z_R^*$.

Proof. Let z_1^* denote the value of an optimal solution of the (MSSP) instance I with a constant number of distinct item weights. Let s be the cardinality of this solution and let $y_1,\dots,y_s$ be the weights of the items packed, in increasing order. Analogously, let $x_1,\dots,x_s$ be the weights of the respective items packed by the heuristic solution for the item set R, again in increasing order. For notational convenience, let $v_{-1} := 0$ throughout the proof.

Observe that $x_{j+k} \geq y_j$ for $j = 1,\dots,s-k$. This ensures that $z_1^* - z_L^H$ is not larger than k times the largest weight in the instance. Hence,

$$z_1^* - z_L^H \leq kv_p \leq kc_{\max} \leq m\tilde{\varepsilon}^{\ell+1}c_{\max} \leq \tilde{\varepsilon}c(M) \leq 2\tilde{\varepsilon}z_R^*,$$

as w.l.o.g. $z_R^* \geq c(M)/2$, since otherwise the application of (FFD) to R would yield the optimal solution.

Now consider the optimal solution for instance R, and let r_j^* be the number of items in R_j packed by this solution for $j = 0,\dots,p$. Furthermore, let z_2^* be the value of the optimal solution of the instance in which there are exactly r_j^* items of weight v_{j-1} for $j = 0,\dots,p$. Observing that the weight of each item in R_j is not smaller than v_{j-1} for $j = 0,\dots,p$, it follows that in this latter solution all the items are packed. Moreover, by definition, $z_1^* \geq z_2^*$, as z_1^* is the solution of an instance defined by a wider item set.

Let $r := \sum_{j=0}^{p} r_j$, and, as above, $y_1,\dots,y_r$ and $x_1,\dots,x_r$ denote the weights (in decreasing order) of the items packed by the solutions of value z_2^* and z_R^*, respectively. One has $y_{i+k} \geq x_i$ for $i = 1,\dots,r-k$, as $r_j^* \leq k$ for $j = 0,\dots,p$. Hence, by the same considerations as above, the relation

$$z_R^* - z_2^* \leq 2\tilde{\varepsilon}z_R^*$$

holds. Therefore we have

$$z_L^H \geq z_1^* - 2\tilde{\varepsilon}z_R^* \geq z_2^* - 2\tilde{\varepsilon}z_R^* \geq (1-2\tilde{\varepsilon})z_R^* - 2\tilde{\varepsilon}z_R^* = (1-4\tilde{\varepsilon})z_R^*.$$

□

We summarize the various steps of Algorithm $PTAS_{CD}$ in Figure 10.4.

Theorem 10.5.2 *Algorithm $PTAS_{CD}$ is a* PTAS *for the capacity dense (MSSP).*

Proof. From Lemma 10.5.1 and inequality (10.30) we have

$$z_L^H \geq (1-4\tilde{\varepsilon})z_R^* \geq (1-4\tilde{\varepsilon})(1-\tilde{\varepsilon})z_L^*.$$

Hence, $z_L^H \geq (1-\varepsilon)z_L^*$ as long as $(1-4\tilde{\varepsilon})(1-\tilde{\varepsilon}) \geq (1-\varepsilon)$, which is satisfied by the choice $\tilde{\varepsilon} := \varepsilon/5$. The running time is clearly polynomial by the discussion above.

□

Algorithm $PTAS_{CD}$:
Initialization:
Given the required accuracy ε, set $\tilde{\varepsilon} := \varepsilon/5$ and partition the item set into sets S and L.
Phase 1:
Form the set of relevant items R.
Phase 2:
If $m \leq 3/\tilde{\varepsilon}^{\ell+1}$ then
    compute the optimal solution for set R by complete enumeration.
else
    group the items in set R,
    solve to optimality instance I,
    construct a feasible solution for item set R from
    the optimal solution obtained.
Phase 3:
Pack the items in S in a greedy way.

Fig. 10.4. A *PTAS* for the capacity dense (MSSP).

Given the required accuracy ε, partition the set M of knapsacks according to capacity intervals into sets $M_1, M_2, \ldots$, where

$$M_\ell := \{i \in M \mid c_i \in]\varepsilon^\ell c_{\max}, \varepsilon^{\ell-1} c_{\max}]\}. \tag{10.34}$$

Let k be the maximum number of nonempty consecutive sets in the partition, i.e.

$$k := \max\{d \mid \exists \ell \text{ such that } M_{\ell-1} = \emptyset, M_{\ell+1} \neq \emptyset, \ldots, M_{\ell+d-1} \neq \emptyset, M_{\ell+d} = \emptyset\},$$

where we define $M_0 := \emptyset$. We say that the (MSSP) instance is *capacity grouped (at ε)* if k is bounded by a constant.

Given a capacity grouped instance, the knapsacks are partitioned as follows. Remove from $M_1, M_2, \ldots$ all the empty sets, and aggregate the remaining sets into groups so that sets with consecutive indices are in the same group. Formally, the sets M_i and M_j, $i < j$, are in the same group if and only if $M_k \neq \emptyset$ for $k = i, \ldots, j$. Note that, by definition of capacity grouped, each group is formed by the union of at most k sets. Hence, the ratio between the minimum and maximum capacity in each group is at most ε^k and we can define large and small items for each group as above. Let the groups, arranged in decreasing order of capacities, be $G_1, G_2, \ldots, G_t$, noting that t is in $O(m)$. The reason for partitioning the set of knapsacks into groups is that the large items for group G_p do not fit into the knapsacks in groups $G_{p+1}, \ldots, G_t$ and are small items for the knapsacks in groups $G_{p-1}, \ldots, G_1$. This allows a separate handling of the large items in each group, as illustrated below.

It is now shown how $PTAS_{CD}$ can be adapted to a polynomial time $(1-\varepsilon)$-approximation algorithm for the capacity grouped case, referred to as H^{ε}_{CG}. We take $PTAS_{CD}$ almost as a black box, even if we use the fact that it first computes a near optimal solution for the large items and then packs the small items greedily.

H^{ε}_{CG} starts from G_t, the group with smallest capacities, and applies $PTAS_{CD}$ to it, considering all items that fit into the largest knapsack in the group. Then, $PTAS_{CD}$ is applied to the knapsacks in group G_{t-1}, considering all the items that fit into the largest knapsack and were not packed before in some knapsack of group G_t. The procedure is iterated, applying $PTAS_{CD}$ to the knapsacks in $G_{t-2}, \ldots, G_1$.

Theorem 10.5.3 *Algorithm H^{ε}_{CG} is a polynomial time $(1-\varepsilon)$-approximation algorithm for the capacity grouped (MSSP).*

Proof. Since H^{ε}_{CG} calls t, i.e. $O(m)$, times $PTAS_{CD}$, its running time is polynomial. We will show by induction that the produced solution is within ε of the optimal one. In fact, we prove that $z^H_p \geq (1-\varepsilon)z^*_p$ for $p = t, t-1, \ldots, 1$, where z^H_p is the value of the heuristic solution produced for knapsacks in $G_t \cup G_{t-1} \cup \ldots \cup G_p$, and z^*_p is the optimal solution value of the instance with these knapsacks only.

The basis of the induction, namely $z^H_t \geq (1-\varepsilon)z^*_t$ follows immediately from Theorem 10.5.2.

Now suppose $z^H_{p+1} \geq (1-\varepsilon)z^*_{p+1}$. Consider the packing of the small items in $PTAS_{CD}$ applied to group G_p.

If all small items are packed, then $z^H_p = z^H_{L_p} + w(S_p)$, where S_p denotes the set of small items for group G_p and $z^H_{L_p}$ is the contribution of the large items for group G_p. For the optimal solution, we can separate the contributions of the large and small items for group G_p, writing $z^*_p = z^*_{L_p} + z^*_{S_p}$. The fact that the large items can be packed only in the knapsacks in G_p and Theorem 10.5.2 imply that $z^H_{L_p} \geq (1-\varepsilon)z^*_{L_p}$, whereas clearly $w(S_p) \geq z^*_{S_p}$. Hence, $z^H_p \geq (1-\varepsilon)z^*_p$.

In the other case, in which not all small items are packed, we have

$$z^H_p \geq (1-\varepsilon)c(G_p) + z^H_{p+1}.$$

Separating the contributions to the optimal solution value of the knapsacks in G_p and in $G_{>p} := G_t \cup \ldots \cup G_{p+1}$, we can write

$$z^*_p = z^*(G_p) + z^*(G_{>p}).$$

Clearly,

$$(1-\varepsilon)c(G_p) \geq (1-\varepsilon)z^*(G_p)$$

and

$$z^H_{p+1} \geq (1-\varepsilon)z^*_{p+1} \geq (1-\varepsilon)z^*(G_{>p}),$$

where the first inequality is the inductive hypothesis above. Hence, $z^H_p \geq (1-\varepsilon)z^*_p$ and the proof is complete. □

Now it is possible to show how to handle general instances of (MSSP). The main idea in [63] is to force an instance to be capacity grouped for the required accuracy ε

by removing some knapsacks, so that the optimal solution values before and after the removal of the knapsacks are close to each other. This yields the *PTAS* for general (MSSP), hereafter called $PTAS_{GEN}$, that uses H^{ε}_{CG} as a black box.

$PTAS_{GEN}$ considers the partitioning of the knapsacks into sets $M_1, M_2, \ldots$ defined as above. Given the required accuracy ε, let $\hat{\varepsilon} := \varepsilon/2$. For each value $q = 1, 2, \ldots, \lceil \frac{1}{\hat{\varepsilon}} \rceil$, $PTAS_{GEN}$ removes all knapsacks in

$$M(q) := M_q \cup M_{q+\lceil \frac{1}{\hat{\varepsilon}} \rceil} \cup M_{q+2\lceil \frac{1}{\hat{\varepsilon}} \rceil} \cup \ldots = \bigcup_{i=0}^{\infty} M_{q+i\lceil \frac{1}{\hat{\varepsilon}} \rceil}.$$

The instance obtained in this way is capacity grouped at $\hat{\varepsilon}$ with $k \leq \lceil \frac{1}{\hat{\varepsilon}} \rceil$, and H^{ε}_{CG} is applied to it. Among all solutions computed for the different values of q, the best one is given as output.

Theorem 10.5.4 *Algorithm $PTAS_{GEN}$ is a* PTAS *for general (MSSP).*

Proof. The running time of $PTAS_{GEN}$ is polynomial as it calls $\lceil \frac{1}{\hat{\varepsilon}} \rceil$ times H^{ε}_{CG}. In order to show that the solution is within ε of the optimum, we show that there exists a value $q \in \{1, 2, \ldots, \lceil \frac{1}{\hat{\varepsilon}} \rceil\}$ such that $z^*(q) \geq (1-\hat{\varepsilon})z^*$, where $z^*(q)$ is the value of the optimal solution after the knapsacks in $M(q)$ have been removed. Then the theorem follows from the fact that

$$z^H(q) \geq (1-\hat{\varepsilon})z^*(q) \geq (1-\hat{\varepsilon})^2 z^* \geq (1-\varepsilon)z^*,$$

where $z^H(q)$ is the value of the solution returned by H^{ε}_{CG}.

Let γ_q be the contribution of the knapsacks in $M(q)$ to the optimal solution value for $q = 1, \ldots, \lceil \frac{1}{\hat{\varepsilon}} \rceil$. Hence, we have

$$z^* = \gamma_1 + \gamma_2 + \ldots + \gamma_{\lceil \frac{1}{\hat{\varepsilon}} \rceil}.$$

Let $q_{\min}$ be such that

$$\gamma_{q_{\min}} = \min_{q=1}^{\lceil \frac{1}{\hat{\varepsilon}} \rceil} \gamma_q.$$

Clearly, $\gamma_{q_{\min}} \leq \hat{\varepsilon} z^*$. Moreover,

$$z^*(q_{\min}) \geq \left(\sum_{q=1}^{\lceil \frac{1}{\hat{\varepsilon}} \rceil} \gamma_q \right) - \gamma_{q_{\min}},$$

as a feasible solution for the instance, where the knapsacks in $M(q)$ have been removed, is given by the packing of the remaining knapsacks in the optimal solution. Overall, we complete the proof by observing

$$z^*(q_{\min}) \geq z^* - \gamma_{q_{\min}} \geq (1-\hat{\varepsilon})z^*.$$

□

10.5.2 A *PTAS* for the Multiple Knapsack Problem

In this section, we will present a description of the *PTAS* for (MKP) by Chekuri and Khanna [79] currently only published in conference proceedings, a generalization of the results in [263] of knapsacks of identical capacities to arbitrary capacities. Their approach is based on the following ideas:

- Approximation algorithms for packing problems, especially *FPTAS*, are often based on rounding instances to get a fixed number of distinct values. In this way in knapsack problems the items with profit greater than εc are usually reduced to an instance with a constant number of different profits. In contrast, to get a *PTAS* for (MKP) only a logarithmic number of different profits and weights is used.
- Using the reduced number of distinct profit values and weights it is shown how to *guess* a near-optimal set, i.e. a feasible set of items which has total profit at least $(1-\varepsilon)z^*$. Guessing a set S of items means to identify in polynomial time a *polynomial number* of item sets among which is set S. The requested item set S can then be found by complete enumeration.
- Another problem is to *find* a feasible packing for the near-optimal set S, obtained by guessing, since this problem is of course also $\mathcal{NP}$-complete. Again, it is guessed to which knapsack among some knapsacks with similar capacities an item is assigned without loosing too much profit. Note that the whole guessing strategy is independent of the number of knapsacks m and their capacities.

We start a more detailed description of the *PTAS* by guessing first a value $\tilde{z}^*$ which is close to the optimum solution value z^*. It follows from

$$\ln(1+\varepsilon) \geq \varepsilon - \frac{1}{2}\varepsilon^2 \tag{10.35}$$

by a rough estimation that $(1+\varepsilon)^{2\ln n/\varepsilon} \geq 2n$ holds. We conclude with

$$p_{\max} \leq z^* \leq n p_{\max},$$

that we can find by guessing in polynomial time a value $\tilde{z}^* := p_{\max}(1+\varepsilon)^i$ with $0 \leq i \leq \lfloor 2\ln n/\varepsilon \rfloor$ such that

$$z^* \geq \tilde{z}^* \geq z^*/(1+\varepsilon) > (1-\varepsilon)z^*.$$

From item set N we get a reduced item set N_1 with $O(\ln n/\varepsilon)$ different profit values as follows: First, discard all items with profit less then or equal to $\varepsilon\tilde{z}^*/n$. This can be done, because the total loss of profit is at most $\varepsilon\tilde{z}^*$. Then round down the profits p_j to the nearest possible value in

$$N_1 := \left\{ \frac{\varepsilon}{n}(1+\varepsilon)^i \tilde{z}^* \mid i = 1,\ldots,2\lfloor \ln n/\varepsilon \rfloor \right\}. \tag{10.36}$$

Using (10.35) more sharply shows that the maximal element of N_1 is larger than $\tilde{z}^*$ if n is large enough. Thus, the $O(\ln n/\varepsilon)$ different profit values of N_1 are at most a factor $(1+\varepsilon)$ smaller than the original profit values. Let z_1^* denote the optimal solution value for instance N_1. Using that all items with profit less then or equal to $\varepsilon\tilde{z}^*/n$ are discarded, we get

$$z_1^* \geq \frac{1}{1+\varepsilon} z^* - \varepsilon\tilde{z}^* \geq (1-2\varepsilon)z^*. \tag{10.37}$$

Therefore, we can work with item set N_1 further on, without loosing too much profit.

Let h be the number of distinct profits in N_1, i.e. h is in $O(\ln n/\varepsilon)$. We partition N_1 into sets $S_1,\ldots,S_h$ with items in each set having the same profit. Let U be the items chosen in some optimal solution for N_1 and set $U_i := S_i \cap U$ for $i = 1,\ldots,h$. By choosing $k_i := \lfloor p(U_i)h/(\varepsilon^2\tilde{z}^*)\rfloor$ for $k_i \in \{0,\ldots,\lfloor h/\varepsilon^2\rfloor\}$ we have

$$k_i\frac{\varepsilon^2\tilde{z}^*}{h} \leq p(U_i) \leq (k_i+1)\frac{\varepsilon^2\tilde{z}^*}{h}. \tag{10.38}$$

For a given k_i sort the items in S_i in increasing order of weights and collect the items until the total profit exceeds $k_i\frac{\varepsilon^2\tilde{z}^*}{h}$. Then for a tuple $(k_1,\ldots,k_h)$ let $U(k_1,\ldots,k_h)$ denote the set of items collected as described above. The following lemma is not difficult to show:

Lemma 10.5.5 *For N_1 there is a tuple $(k_1,\ldots,k_h)$ with $k_i \in \{0,\ldots,\lfloor h/\varepsilon^2\rfloor\}$ for $i = 1,\ldots,h$ such that the corresponding item set $U(k_1,\ldots,k_h)$ is feasible with total profit at least $(1-\varepsilon)\tilde{z}^*$.*

For guessing an appropriate tuple for a *PTAS* we have to assure that the number of possible tuples $(k_1,\ldots,k_h)$ is polynomial. Since $\sum_{i=1}^{h} k_i \leq \lceil h/\varepsilon^2\rceil$, we can apply two simple combinatorial facts, collected in the following lemma.

Lemma 10.5.6 *Let f be the number of g-tuples of nonnegative integers such that the sum of tuple coordinates is less than or equal to d. Then $f = \binom{d+g}{g}$. If $d+g \leq \alpha g$, then $f = O(\alpha^{\alpha g})$.*

Proof. If we mark g points $p_1,\ldots,p_g$ among $d+g$ consecutive points, then the number of unmarked points between the markers p_i and p_{i+1}, $i = 1,\ldots,g-1$, corresponds to the i-th coordinate of the g-tuple. This implies the first equation. The second inequality follows directly from Stirling's approximation for the factorial. □

If we apply Lemma 10.5.6 for $g := h$ and $d := \lceil h/\varepsilon^2\rceil$, we get that the number of possible tuples $(k_1,\ldots,k_h)$ is $\binom{\lceil h/\varepsilon^2\rceil+h}{h}$ and thus is in $O\left((1+1/\varepsilon^2)^{(1+1/\varepsilon^2)h}\right)$ which is equivalent to $O\left((1+1/\varepsilon^2)^{O(\ln(n^{1/\varepsilon^3}))}\right)$ or $O\left(n^{O(1/\varepsilon^3)}\right)$.

Thus, we can assume in the remainder that we have guessed an item set $N_2 := U(k_1,\ldots,k_h)$ with the properties of Lemma 10.5.5 and having at most $O(\ln n/\varepsilon)$ different profit values. In order to reduce also the number of different weights to a logarithmic number, we apply for each item set$R$ of N_2 with identical profits the *shifting technique* already being used in Section 10.5.1 for (MSSP).

Set $p := \lfloor 1/\varepsilon \rfloor$ and let for $r := |R|$ the values k and q be such that $r = pk + q$ and $1 \leq q \leq k$. Now assume the items in R to be sorted in increasing order of weights and split R into p groups $R_1,\ldots,R_p$ containing k items and group R_{p+1} containing q items. The items in R_{p+1} are discarded and for each $i \leq p$ the weight of every item in R_i is increased to the weight of the smallest item in R_{i+1}. Thus, R is modified into a set with at most p distinct weights. It can be easily seen that the new instance with modified R is still feasible. Since $q \leq r/p \leq \varepsilon r$, we loose at most $\varepsilon p(R)$ of the profit of R. Altogether, doing this for all sets of identical profits, we obtain a new item set N_3 with $O(\ln n/\varepsilon^2)$ different weights, loosing at most an ε fraction of the total profit. Summarizing we have:

Lemma 10.5.7 *For (MKP) with item set N we can find in polynomial time item set N_3 with the following properties:*

- *N_3 has only $O(\ln n/\varepsilon)$ different profit values and $O(\ln n/\varepsilon^2)$ different weights.*
- *N_3 is feasible with $p(N_3) \geq (1 - O(\varepsilon))z^*$.*
- *N_3 can transformed in polynomial time in a feasible subset N' of N with $p(N') \geq (1 - O(\varepsilon))z^*$.*

Assume the knapsacks to be sorted in increasing order of their capacity according to (10.26). Now we aggregate the knapsacks into *groups* $G_0, G_1, \ldots, G_\ell$ such that group G_i consists of all knapsacks having capacities c with

$$(1+\varepsilon)^i \leq c < (1+\varepsilon)^{i+1}. \tag{10.39}$$

Set $g_i := |G_i|$. A group G_i is called a *small group* if $g_i \leq 1/\varepsilon$, otherwise it is called *large*. Chekuri and Khanna show that it is possible to retain the $3\ln(1/\varepsilon)/\varepsilon$ small groups with largest capacities and discard the remaining small groups. Then we guess for each knapsack of the remaining small groups, the $1/\varepsilon$ most profitable items which are packed in the optimal solution. The number of guesses needed is in $n^{O(\ln(1/\varepsilon)/\varepsilon^3)}$. Since this part is mainly technical, we omit further details on the small groups, and refer the reader to [79]. Assume in the following that all groups are large.

We say that an item is *packed as large* if its weight is at least ε times the capacity of the knapsack to which it is assigned, otherwise it is said to be *packed as small*. Let G_i and G_k be the smallest and the largest group, respectively, in which a certain item j can be packed as large. Then, we have

$$\varepsilon(1+\varepsilon)^k \leq w_j < (1+\varepsilon)^i.$$

Using the fact that $\ln(1+\varepsilon) \geq \varepsilon/2$, we conclude that an item can be packed as large in at most $f := \lceil \frac{2\ln(1/\varepsilon)}{\varepsilon} \rceil$ different groups. Denote by Y_i the items of N_3 with the$i$-th weight value and set $y_i := |Y_i|$. The next step is to guess all items which are packed as large and to guess also the groups to which they are assigned. This can be roughly described as follows:

- Denote by y_i^* the number of items of Y_i packed as large in some optimal solution. If $y_i^* \leq \varepsilon y_i$, we can discard these items. Otherwise, a number $\tilde{y}_i^*$ such that $y_i^*(1-\varepsilon) \leq \tilde{y}_i^* \leq y_i^*$ can be found in $O(1/\varepsilon)$ guesses.
- Partition the $\tilde{y}_i^*$ items into f_i groups ($f_i \leq f$) where f_i is the number of groups in which the items of Y_i can be packed as large. This can be done in $n^{O(\ln(1/\varepsilon)/\varepsilon^3)}$ guesses.
- Doing this for all weights requires $n^{O(\ln(1/\varepsilon)/\varepsilon^5)}$ guesses using the fact that the number of distinct weights is in $O(\ln n/\varepsilon^2)$.

Assume we have determined those items which are packed as large in group G_i. Denote these items by V_i. Then group G_i consists of g_i knapsacks with capacities in the range $[(1+\varepsilon)^i, (1+\varepsilon)^{i+1}[$. All items in V_i have weights in the range $]\varepsilon(1+\varepsilon)^i, (1+\varepsilon)^{i+1}[$. The next step is to find a packing for V_i. This is done in the following way:

- Sort the items of V_i with weight greater than $(1+\varepsilon)^i$ in increasing order. Pack each of these items into the smallest knapsack from G_i in which it fits.
- Disregard all knapsacks used in the previous step, since no other item of V_i can be packed into them.
- Assign the remaining items of V_i to group G_i by adapting a *PTAS* for bin packing (see e.g. [142]), possibly using at most εg_i extra knapsacks of capacity c.

The extra knapsacks are eliminated later in the algorithm by picking the "most profitable" among them and discarding the items packed in the remaining extra knapsacks. Note that in order to use extra knapsacks, the value εg_i must be at least 1. This is the reason to distinguish between small and large groups.

The remaining items are assigned to the knapsacks by a generalized assignment problem (GAP). This (GAP) is formulated such that all remaining items are assigned only to knapsacks in which they will be packed as small and that the capacity constraints are not violated. Since all remaining items are packed as small, the analysis of Shmoys and Tardos [435] for (GAP) shows that the fractional solution can be transformed into an integral assignment of the remaining items by using for each group G_i at most $2\varepsilon g_i$ extra knapsacks of capacity $(1+\varepsilon)^i$.

Including the items packed as large, there are altogether at most $3\varepsilon g_i$ extra knapsacks of capacity $(1+\varepsilon)^i$ for each group G_i. Denote the set of these knapsacks by E_i and

let P_i be the total profit of all items assigned to $G_i \cup E_i$. The g_i most profitable knapsacks in $G_i \cup E_i$ have a total profit of at least $P_i/(1+3\varepsilon) \geq (1-3\varepsilon)P_i$. Thus, by removing the other knapsacks for each group G_i, no significant profit is lost and we get a feasible assignment. This finishes the description of the *PTAS* by Chekuri and Khanna [79] for (MKP).

Theorem 10.5.8 *There is a* PTAS *for the multiple knapsack problem.*

10.6 Variants of the Multiple Knapsack Problem

In the last section of this chapter we will finally present two variations of (MKP) which have been recently investigated: the multiple knapsack problem with assignment restrictions and the class-constrained multiple knapsack problem.

10.6.1 The Multiple Knapsack Problem with Assignment Restrictions

The *multiple knapsack problem with assignment restrictions* (MKPAR) is a variant of (MKP) where each item can only be assigned to a specified subset of the knapsacks. In general, we are given an instance of (MKP), i.e a set of items $N = \{1,\ldots,n\}$ with profits p_j and weights w_j, $j = 1,\ldots,n$, and a set of knapsacks $M = \{1,\ldots,m\}$ with capacities c_i, $i = 1,\ldots,m$. In addition, for each item j a set $A_j \subseteq M$ of knapsacks, into which item j can be packed, is given. Equivalenty, we can define sets B_i containing the items that can be assigned to knapsack i. The assignment restrictions can be illustrated by a bipartite graph where the two groups of vertices correspond to the sets N and M and there is an edge between vertex $j \in N$ and $i \in M$ if and only if $j \in B_i$. Analogously to (MKP), the objective is to select a "feasible" subset of N such that the total profit of this subset is maximized.

There are only a few results published about (MKPAR). Dawande et al. [100] introduced (MKPAR) for the multiple subset sum case, i.e. for $p_j = w_j$, denoted by MSSPAR. According to Theorem 10.4.1 MSSPAR does not admit an *FPTAS* unless $\mathcal{P} = \mathcal{NP}$. Their investigation of MSSPAR was motivated from an application in inventory matching in the steel industry. They presented a simple greedy-type heuristic with relative performance guarantee $1/3$ and two $1/2$-approximation algorithms. One is based on solving classical knapsack problems successively, the other one is based on rounding the LP-relaxation of the integer programming formulation. All these bounds are tight.

10.6.2 The Class-Constrained Multiple Knapsack Problem

The *class-constrained multiple knapsack problem* (CCMKP) extends (MKP) in the way that each item is characterized not only by profit and weight, but also by a

color or *class* in the sense that items of different colors have to be put into different *compartments* of the knapsacks. Formally, each item j, $j = 1,\ldots,n$, of an instance of (CCMKP) has profit p_j, weight w_j and a color (class) $h \in \{1,\ldots,f\}$. Each knapsack i has capacity c_i and d_i *compartments*. A subset of the items is feasible if the items of this subset can be assigned to the knapsacks without exceeding the capacities and each knapsack i contains no more than d_i items of different colors. The objective is to select a feasible subset with maximal total profit. Class-constrained multiple knapsack problems were introduced by Shachnai and Tamir [419] in 1998. They listed several important applications for (CCMKP), including storage management for multimedia systems and production planning problems.

Shachnai and Tamir [420] investigated a special case of (CCMKP) in which all profits and weight are equal to one. In this case the objective turns into maximizing the number of items which can be assigned to the knapsacks. They showed that this special case, denoted by UCCMKP, is still $\mathcal{NP}$-hard and presented an approximation algorithm with relative performance guarantee of $\delta/(\delta+1)$, when $d_i \geq \delta$ for $i = 1,\ldots,m$. Golubchik et al. [188] derived a tighter bound of $1 - 1/(1+\sqrt{\delta})^2$ for this algorithm, with a matching lower bound for any algorithm for UCCMKP and developed a *PTAS*. They extended the $\mathcal{NP}$-hardness proof of Shachnai and Tamir by showing that UCCMKP is strongly $\mathcal{NP}$-hard even for identical knapsacks which excludes the existence of an *FPTAS* unless $\mathcal{P} = \mathcal{NP}$ holds. Let us mention that Shachnai and Tamir [422] gave also tight bounds for the *on-line* version of UCCMKP.

In 2001, Shachnai and Tamir [421] presented a *PTAS* for general (CCMKP), assuming that f, the number of distinct colors, is a constant. Their proof is partially based on the *PTAS* for (MKP) by Chekuri and Khanna [79] described in Section 10.5. In this paper they considered also the *class-constrained knapsack problem* (CCKP) which is the restriction of (CCMKP) to one knapsack. Note that (CCKP) is a generalization of the k-item knapsack problem (kKP), treated in Section 9.7, when the colors of the items are all distinct and the knapsack accepts at most k different colors. They presented an *FPTAS* for (CCKP) with running time in $O(f^2 n^2 (1+\frac{f}{d\varepsilon}) \cdot \frac{1}{\varepsilon})$ where d is the number of items with different colors accepted by the knapsack. As in the *FPTAS* for (KP) their algorithm is based on dynamic programming and scaling of the profit values.

11. The Multiple-Choice Knapsack Problem

The *multiple-choice knapsack problem* (MCKP) is a generalization of the ordinary knapsack problem, where the set of items is partitioned into classes. The binary choice of taking an item is replaced by the selection of exactly one item out of each class of items. In Section 7.1 we already noticed that a (BKP) can be formulated as a (MCKP), and indeed the (MCKP) model is one of the most flexible knapsack models. (MCKP) is also denoted as *knapsack problem with generalized upper bound constraints* or for short *knapsack problem with GUB*.

11.1 Introduction

In order to define the problem formally, consider m mutually disjoint classes $N_1,\dots,N_m$ of *items* to be packed into a knapsack of *capacity* c. Each item $j \in N_i$ has a profit p_{ij} and a weight w_{ij}, and the problem is to choose exactly one item from each class such that the profit sum is maximized without exceeding the capacity c in the corresponding weight sum. If we introduce the binary variables x_{ij} which take on value 1 if and only if item j is chosen in class N_i, the problem is formulated as:

$$\text{(MCKP)} \qquad \text{maximize} \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij}$$

$$\text{subject to} \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \le c, \tag{11.1}$$

$$\sum_{j \in N_i} x_{ij} = 1, \quad i = 1,\dots,m, \tag{11.2}$$

$$x_{ij} \in \{0,1\}, \quad i = 1,\dots,m, \quad j \in N_i. \tag{11.3}$$

If we relax the integrality constraint (11.3) on x_{ij} we obtain the *linear multiple-choice knapsack problem* C(MCKP).

In the sequel we will assume that all coefficients p_{ij}, w_{ij}, and c are nonnegative integers, with class N_i having *size* n_i. The total number of items is $n := \sum_{i=1}^{m} n_i$. Negative coefficients p_{ij}, w_{ij} in (MCKP) may be handled by adding the constant

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

$\overline{p}_i := -\min_{j\in N_i} p_{ij}$ to all the profits in class N_i, and adding $\overline{w}_i := -\min_{j\in N_i} w_{ij}$ to all weights in N_i as well as to the capacity c. To avoid unsolvable or trivial situations we assume that

$$\sum_{i=1}^{m} \min_{j\in N_i} w_{ij} \le c < \sum_{i=1}^{m} \max_{j\in N_i} w_{ij}, \tag{11.4}$$

and moreover we assume that every item $j \in N_i$ satisfies

$$w_{ij} + \sum_{h=1,\ldots,m,h\neq i} \min_{k\in N_h} w_{hk} \le c, \tag{11.5}$$

as otherwise no feasible solution exists when the item is chosen, and hence it may be discarded.

The (MCKP) in minimization form may be transformed into an equivalent maximization problem (MCKP) by finding for each class N_i the values $\overline{p}_i := \max_{j\in N_i} p_{ij}$ and $\overline{w}_i := \max_{j\in N_i} w_{ij}$, and by setting $\tilde{p}_{ij} := \overline{p}_i - p_{ij}$ and $\tilde{w}_{ij} := \overline{w}_i - w_{ij}$ for $j \in N_i$ and $\tilde{c} := \sum_{i=1}^{m} \overline{w}_i - c$. Then the maximization problem is defined in $\tilde{p}, \tilde{w}$ and $\tilde{c}$.

If the multiple-choice constraints (11.2) are replaced by $\sum_{j\in N_i} x_{ij} \le 1$, as considered in e.g. Johnson and Padberg [254], then this problem can be transformed into the equality form by adding a dummy item $(p_{i,n_i+1}, w_{i,n_i+1}) := (0,0)$ to each class N_i.

(MCKP) is $\mathcal{NP}$-hard, which easily can be shown by reduction from the *knapsack problem*: Given an instance of (KP) with n profits p_j, weights w_j and capacity c, construct an instance of (MCKP) by introducing $m := n$ classes each class i having two items $(p_{i1}, w_{i1}) := (0,0)$ respectively $(p_{i2}, w_{i2}) := (p_j, w_j)$, while the capacity of the new problem is c.

The (MCKP) problem has a wide range of applications: Nauss [355] mentions *capital budgeting* and transformation of *nonlinear knapsack problems* (see Section 13.3.1) to (MCKP) as possible applications, while Sinha and Zoltners [436] propose (MCKP) used for *menu planning* or to determine which components should be linked in series in order to maximize fault tolerance. Witzgall [485] proposes (MCKP) used to accelerate ordinary LP/GUB problems by the dual simplex algorithm. Moreover (MCKP) appears by Lagrangian relaxation of several integer programming problems, as described by Fisher [147]. The (MCKP) has been used for solving *generalized assignment problems* by Barcia and Jörnsten [26]. Several applications of (MCKP) appear in *VLSI design*. Alvarez, Levner and Mosse [344] consider the problem of limiting power consumption in VLSI circuits, while de Leone, Jain and Straus [102] consider problems in high-level synthesis of VLSI circuits.

In Section 11.2 we will present some dominance relations and derive a number of upper bounds. Reduction of classes is discussed in Section 11.3, and these results are applied in the branch-and-bound part presented in Section 11.4 and 11.5. Some reduction rules to fathom states in an enumerative algorithm are given in Section 11.6, followed by a presentation of hybrid algorithms and expanding-core algorithms in Section 11.7. Computational results are given in Section 11.8. We conclude this chapter with some heuristic solution methods in Section 11.9.

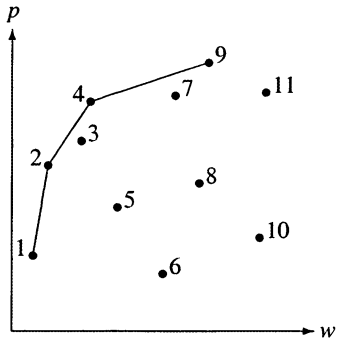


Fig. 11.1. The concept of dominance. E.g. item 5 is dominated by item 2, item 3 is LP-dominated by items 2 and 4. The set of undominated items is $\{1,2,3,4,7,9\}$. The set of LP-extreme items is $R_i = \{1,2,4,9\}$. Notice that R_i forms the upper left convex hull of N_i.

11.2 Dominance and Upper Bounds

The concept of *dominance* plays a significant role in the solution of (MCKP), since dominance makes it possible to delete some items which never will be chosen in an optimal solution. When dealing with the integer-optimal solution "simple" dominance is used, while a stronger LP-dominance may be used when dealing with the linear relaxation C(MCKP).

Definition 11.2.1 *If two items j and k in the same class N_i satisfy*

$$w_{ij} \leq w_{ik} \quad \text{and} \quad p_{ij} \geq p_{ik}, \tag{11.6}$$

then we say that item k is dominated *by item j. Similarly, if some items $j,k,\ell \in N_i$ with $w_{ij} < w_{ik} < w_{i\ell}$ and $p_{ij} < p_{ik} < p_{i\ell}$ satisfy*

$$\frac{p_{i\ell} - p_{ik}}{w_{i\ell} - w_{ik}} \geq \frac{p_{ik} - p_{ij}}{w_{ik} - w_{ij}}, \tag{11.7}$$

then we say that item k is LP-dominated *by items j and ℓ.*

The two concepts of dominance are illustrated in Figure 11.1. Sinha and Zoltners [436] showed the following straightforward proposition, which makes it possible to reduce a-priori the size of an instance:

Proposition 11.2.2 *Given two items $j,k \in N_i$. If item k is dominated by item j then an optimal solution to (MCKP) with $x_{ik} = 0$ exists. If an item $k \in N_i$ is LP-dominated by two items $j,\ell \in N_i$ then an optimal solution to C(MCKP) with $x_{ik} = 0$ exists.*

The simplest upper bound is obtained from the LP-relaxation C(MCKP)

$$U_1 := z(C(\text{MCKP})). \tag{11.8}$$

In this case dominated and LP-dominated items may be fathomed, which is easily obtained by ordering the items in each class N_i according to increasing weights, and successively testing the items according to criteria (11.6) and (11.7). For each class N_i the reduction takes $O(n_i \log n_i)$ time due to the sorting. The remaining items R_i are called the *LP-extreme* items, and they form the upper convex hull of N_i, as illustrated in Figure 11.1. The size of set R_i will be denoted r_i. We will assume the ordering $w_{i1} < w_{i2} < \ldots < w_{i,r_i}$ in R_i.

The LP-bound $z(C(\text{MCKP}))$ can now be found by using the greedy algorithm outlined in Figure 11.2.

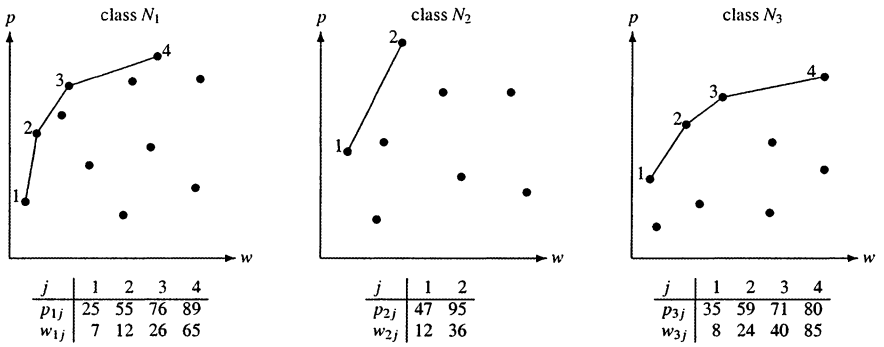
Algorithm MCKP-Greedy:

1. For each class N_i sort the items according to increasing weights and derive R_i. The following indices refer to items in R_i.
2. Construct an instance of (KP) by setting
   $\tilde{p}_{ij} := p_{ij} - p_{i,j-1}$, $\tilde{w}_{ij} := w_{ij} - w_{i,j-1}$ for each class R_i and $j = 2, \ldots, r_i$.
   The residual capacity is $\bar{c} := c - \sum_{i=1}^{m} w_{i1}$.
3. Sort the items according to decreasing incremental efficiencies $\tilde{e}_{ij} := \tilde{p}_{ij}/\tilde{w}_{ij}$. With each value of $\tilde{e}_{ij}$ we associate the original indices i, j during the sorting.
4. Use algorithm Greedy to fill the knapsack up to capacity $\bar{c}$.
   Initialize $z := \sum_{i=1}^{m} p_{i1}$.
   Each time we insert an item with indices i, j
   set $z := z + \tilde{p}_{ij}$, $\bar{c} := \bar{c} - \tilde{w}_{ij}$, $x_{ij} := 1$, $x_{i,j-1} := 0$.
5. Assume that the first item which exceeds the capacity is item $t \in N_s$.
   Set $x_{st} := \bar{c}/\tilde{w}_{st}$, $x_{s,t-1} := 1 - x_{st}$, $z := z + \tilde{p}_{st} x_{st}$.
6. Return the LP-solution x with value z.

Fig. 11.2. The MCKP-Greedy algorithm transforms the (MCKP) into a kind of ordinary knapsack problem, which then can be solved to LP-optimality by the straightforward greedy algorithm from Section 2.1.

The greedy algorithm is based on the transformation principles presented in Zemel [497]. In Step 2, the incremental profit $\tilde{p}_{ij}$ is a measure of how much we gain if we choose item j *instead* of item $j-1$ in class R_i. The incremental weight $\tilde{w}_{ij}$ has a similar interpretation. In Step 5, the item $t \in N_s$ is denoted the *split item*, and the class N_s is denoted the *split class*.

The running time of algorithm MCKP-Greedy is $O(\sum_{i=1}^{m} n_i \log n_i + n \log n)$ where the first term comes from the reduction in Step 1, and the latter term comes from the sorting in Step 3. The running time may be decreased to $O(\sum_{i=1}^{m} n_i \log n_i)$ by observing that we may use the partitioning algorithm Split from Section 3.1 without any sorting to find the greedy solution in Step 4.



j	1	2	3	4
p_{1j}	25	55	76	89
w_{1j}	7	12	26	65

j	1	2
p_{2j}	47	95
w_{2j}	12	36

j	1	2	3	4
p_{3j}	35	59	71	80
w_{3j}	8	24	40	85

Fig. 11.3. Illustration of the MCKP-Greedy algorithm.

Example: To illustrate the MCKP-Greedy algorithm consider the instance given in Figure 11.3 with capacity $c = 100$.

After removal of the LP-dominated items and renumbering the remaining items we have $R_1 = \{1,2,3,4\}, R_2 = \{1,2\}, R_3 = \{1,2,3,4\}$. The incremental efficiencies in sorted order become $\{e_{12} = \frac{30}{5}, e_{22} = \frac{48}{24}, e_{13} = \frac{21}{14}, e_{32} = \frac{24}{16}, e_{33} = \frac{12}{16}, e_{14} = \frac{13}{39}, e_{34} = \frac{9}{45}\}$. Choosing the smallest weights in each class the residual capacity becomes $\bar{c} = c - w_{11} - w_{21} - w_{31} = 73$. The greedy algorithm first chooses item $(1,2)$ reducing the capacity to $\bar{c} = 68$, then it chooses item $(2,2)$ getting $\bar{c} = 44$, item $(1,3)$ with $\bar{c} = 30$, and finally item $(3,2)$ getting $\bar{c} = 14$. The next item $(3,3)$ does not fit into the knapsack hence class $s := 3$ is the split class, and item 3 is the split item. The optimal solution becomes $x_{13} = 1$, $x_{22} = 1$ and finally $x_{33} = \frac{14}{16}$ and $x_{32} = \frac{2}{16}$. The value of the solution is $z = 76 + 95 + 59 + 12\frac{14}{16} = 240\frac{1}{2}$.

As a consequence of the greedy algorithm we have:

Corollary 11.2.3 *An optimal solution x^{LP} to $C(MCKP)$ satisfies the following:*

1) x^{LP} has at most two fractional variables.

2) If x^{LP} has two fractional variables they must be adjacent variables in the (sorted) class R_s.

3) If x^{LP} has no fractional variables, then the split solution is an optimal solution to (MCKP).

The *LP-optimal choices* t_i obtained by the greedy algorithm are the variables for which $x_{it_i} = 1$. In the split class N_s we have two *fractional variables* x_{st_s} and $x_{st'_s}$ of which possibly $x_{st'_s} = 0$. A greedy integer solution to (MCKP) may be constructed by choosing the LP-optimal items, i.e. by setting $x_{it_i} = 1$ for $i = 1,\ldots,m$ and $x_{ij} = 0$ for $i = 1,\ldots,m$, $j \neq t_i$. The solution will be denoted the *split solution* and the corresponding weight and profit sums are $\hat{p} = \sum_{i=1}^{m} p_{it_i}$ and $\hat{w} = \sum_{i=1}^{m} w_{it_i}$, respectively.

The attentive reader may have noticed that the determination of the LP-extreme points is closely related to the problem of finding the *convex hull* of a set of points in the plane. Indeed, using the *prune and search method* by Kirkpatrick and Seidel [270] we could find the set R_i in time $O(n_i \log h_i)$, where h_i is the number of vertices of the convex hull. The linear time algorithm for solving the LP-relaxed problem to be presented in the next section is conceptually similar to the Kirkpatrick and Seidel algorithm for deriving a convex hull, since it repeatedly throws away a constant factor of the remaining points until the upper portion of the convex hull is determined.

11.2.1 Linear Time Algorithms for the LP-Relaxed Problem

Dyer [122] and Zemel [498] independently developed algorithms for C(MCKP), running in $O(n)$ time, which do not use the time-consuming preprocessing of classes N_i to R_i. Both algorithms are based on the convexity of the LP-dual problem to (MCKP), which makes it possible to *pair* the dual line segments, so that at each iteration at least $1/6$ of the line segments are deleted. In the following we will present a primal algorithm which should intuitively be more appealing and which can be seen as a generalization of algorithm Split in Section 3.1. This variant of the algorithm was originally presented in Pisinger and Toth [394]. The algorithm relies on the property that in order to solve C(MCKP), it is sufficient to find the *optimal slope*, i.e. the incremental efficiency of the last item added in the algorithm MCKP-Greedy.

Proposition 11.2.4 *If the optimal slope α^* is known, then the corresponding optimal solution to C(MCKP) can be determined in $O(n)$ time.*

Proof. We construct the optimal solution as follows: For each class N_i we determine $M_i(\alpha^*)$, where $M_i(\alpha)$ is the set of most extreme items in direction $(-\alpha, 1)$ given by

$$M_i(\alpha) := \arg\max_{j \in N_i} \{p_{ij} - \alpha w_{ij}\}. \tag{11.9}$$

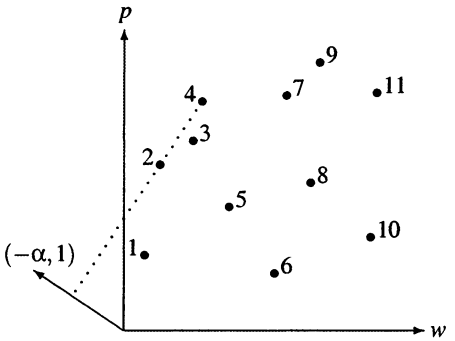


Fig. 11.4. Projection of $(w_{ij}, p_{ij}) \in N_i$ on $(-\alpha, 1)$. Here we have $M_i(\alpha) = \{2,4\}$ and $a_i = 2$, $b_i = 4$.

As each set M_i may contain more than one item, we define a_i as the extreme item in M_i having the smallest weight, and b_i as the item in M_i having the largest weight

$$a_i := \arg \min_{j \in M_i(\alpha^*)} \{w_{ij}\}, \qquad b_i := \arg \max_{j \in M_i(\alpha^*)} \{w_{ij}\}. \tag{11.10}$$

This is graphically illustrated in Figure 11.4. Now, choose item a_i in each class N_i by setting $x_{ia_i} := 1$ for each class $i = 1, \ldots, m$. Set the residual capacity $\bar{c} := c - \sum_{i=1,\ldots,m} w_{ia_i}$. Running through the classes in arbitrary order exchange item a_i by item b_i in class N_i by setting $x_{ia_i} := 0$, $x_{ib_i} := 1$ and reducing the residual capacity $\bar{c} := \bar{c} + w_{a_i} - w_{b_i}$, until $\bar{c} + w_{a_i} - w_{b_i} < 0$ for some class i. Set $x_{b_i} := \bar{c}/(w_{b_i} - w_{a_i})$ and $x_{a_i} := 1 - x_{b_i}$.

To see that the obtained solution is LP-optimal we notice that $a_i, b_i \in R_i$ as obviously both items are LP-extreme. The optimal slope α^* corresponds to the efficiency of the split item in Step 5 of algorithm MCKP-Greedy, and we have $e_{ia_i} > \alpha^*$ and $e_{ib_i} = \alpha^*$. □

Algorithm Dyer-Zemel:

repeat forever

1. for all classes N_i
   Pair the items two by two as (ij, ik).
   Order each pair such that $w_{ij} \leq w_{ik}$ breaking ties such that $p_{ij} \geq p_{ik}$.
   if item j dominates item k in class N_i then
   Delete item k from N_i and pair item j with another item from N_i.
   Continue this process until all items in N_i have been paired
   (leaving one item unpaired if the number of items is odd).
2. for all classes N_i
   if the class has only one item j left then
   Decrease the capacity $c := c - w_{ij}$, fathom class N_i.
3. for all pairs (ij, ik)
   Derive the slope $\alpha_{ijk} := \frac{p_{ik} - p_{ij}}{w_{ik} - w_{ij}}$.
   Let α be the median of the slopes $\{\alpha_{ijk}\}$.
4. for $i = 1, \ldots, m$
   Derive $M_i(\alpha)$ and a_i, b_i according to (11.9) and (11.10).
5. if α is optimal according to (11.11), i.e. if $\sum_{i=1}^{m} w_{ia_i} \leq c < \sum_{i=1}^{m} w_{ib_i}$, then
   α is the optimal slope α^*. Stop.
6. if $\sum_{i=1}^{m} w_{ia_i} \geq c$ then
   for all pairs (ij, ik) with $\alpha_{ijk} \leq \alpha$ delete item k.
7. if $\sum_{i=1}^{m} w_{ib_i} < c$ then
   for all pairs with $\alpha_{ijk} \geq \alpha$ delete item j.

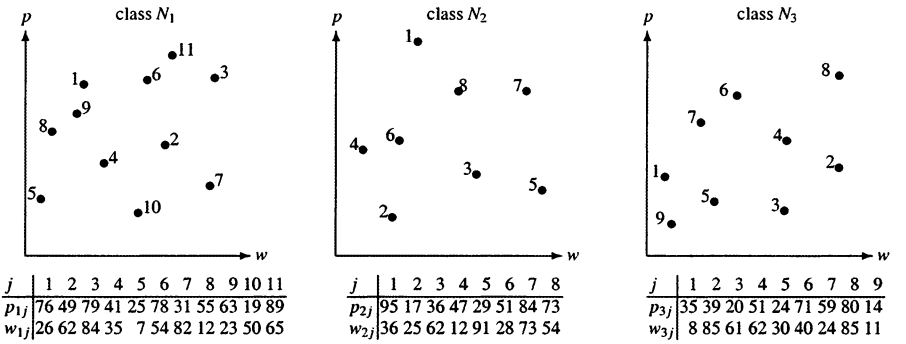
Fig. 11.5. Dyer-Zemel is a partitioning algorithm for finding the optimal slope α^*.

Due to Proposition 11.2.4 we may restate C(MCKP) as finding a slope α such that

$$\sum_{i=1}^{m} w_{ia_i} \leq c < \sum_{i=1}^{m} w_{ib_i}, \tag{11.11}$$

when we determine the extreme items in equations (11.9) and (11.10). The optimal slope α^* may be found by the partitioning algorithm Dyer-Zemel outlined in Figure 11.5.

In Step 6 of Dyer-Zemel we know that the chosen α is too small according to the greedy algorithm, thus the validity of deleting k can be stated as follows: If $j,k \in R_i$, then since $\alpha_{ijk} \leq \alpha$ we may delete k. If $k \notin R_i$ then k cannot be in an optimal solution of C(MCKP) and may be deleted. Thus the final case is $j \notin R_i$ while $k \in R_i$. Let j' be the predecessor of k in R_i. Since $\alpha_{ij'k} < \alpha_{ijk} \leq \alpha$ we may delete k. The validity of Step 7 is checked in a similar way.



j	1	2	3	4	5	6	7	8	9	10	11
p_{1j}	76	49	79	41	25	78	31	55	63	19	89
w_{1j}	26	62	84	35	7	54	82	12	23	50	65

j	1	2	3	4	5	6	7	8
p_{2j}	95	17	36	47	29	51	84	73
w_{2j}	36	25	62	12	91	28	73	54

j	1	2	3	4	5	6	7	8	9
p_{3j}	35	39	20	51	24	71	59	80	14
w_{3j}	8	85	61	62	30	40	24	85	11

Fig. 11.6. Illustration of the Dyer-Zemel algorithm.

Example: To illustrate how the Dyer-Zemel algorithm works, consider the instance given in Figure 11.6 with capacity $c = 100$.

Initially we pair the items in the classes. We choose to pair the items in the order we meet them. In N_1 we pair items (1,2) but 2 is dominated so we pair (1,3). Next we pair (4,5) then (6,7) but since 7 is dominated we pair (6,8). Finally we pair (9,10) noticing that 10 is dominated and hence pairing (9,11). In N_2 we pair (1,2), then (3,4) but since 3 is dominated we try (4,5) but again 5 is dominated and hence we end with (4,6). Finally (7,8) are paired, but 7 is dominated by 8, hence we get the singleton 8. In N_3 we pair (1,2), (3,4), (5,6) and finally (7,8). In this set we have the singleton 9. The slopes are $\alpha_{1,1,3} = \frac{3}{58}$, $\alpha_{1,5,4} = \frac{16}{28}$, $\alpha_{1,8,6} = \frac{23}{42}$, $\alpha_{1,9,11} = \frac{26}{42}$, $\alpha_{2,1,2} = \frac{78}{11}$, $\alpha_{2,4,6} = \frac{4}{16}$, $\alpha_{3,1,2} = \frac{4}{77}$, $\alpha_{3,3,4} = \frac{31}{1}$, $\alpha_{3,5,6} = \frac{47}{10}$, $\alpha_{3,7,8} = \frac{21}{61}$. The median of these is $\alpha = \frac{16}{28}$. We find $M_1(\alpha) = 1$, $M_2(\alpha) = 1$, $M_3(\alpha) = 6$. Since the weight sum is $26 + 36 + 40 \geq c$ we delete items (1,3), (1,4), (1,6), (2,6), (3,2), (3,8) and repeat the process. □

Theorem 11.2.5 *The time complexity of the* Dyer-Zemel *algorithm is* $O(n)$.

Proof. Assume that all items and all classes are represented as lists, such that deletions can be made in $O(1)$. At any stage, n_i refers to the current number of items in class N_i and m is the current number of classes. We will use the terminology (ij,ik) to denote a pair of items $j,k \in N_i$. Notice that each iteration of Steps 1-7 can be performed in time linear in the current number of items.

There are $\sum_{i=1}^{m} \lfloor n_i/2 \rfloor$ pairs of items (ij,ik). Since α is the median of $\{\alpha_{ijk}\}$, half of the pairs will satisfy the criteria in Steps 6 or 7, and thus one item from these pairs will be deleted, i.e. at least $\frac{1}{2}\sum_{i=1}^{m} \lfloor n_i/2 \rfloor$ items are deleted out of $n = \sum_{i=1}^{m} n_i \leq \sum_{i=1}^{m} (2\lfloor n_i/2 \rfloor + 1)$. Since $\lfloor n_i/2 \rfloor \geq 1$, each iteration deletes at least

$$\frac{\frac{1}{2}\sum_{i=1}^{m} \lfloor n_i/2 \rfloor}{\sum_{i=1}^{m} (2\lfloor n_i/2 \rfloor + 1)} \geq \frac{\sum_{i=1}^{m} \lfloor n_i/2 \rfloor}{2m + 4\sum_{i=1}^{m} \lfloor n_i/2 \rfloor} \geq \frac{1}{6}, \tag{11.12}$$

of the items. The running time now becomes

$$O\left(n + \frac{5}{6}n + \left(\frac{5}{6}\right)^2 n + \left(\frac{5}{6}\right)^3 n + \ldots\right) = O\left(\frac{n}{1-\frac{5}{6}}\right) = O(n), \tag{11.13}$$

which shows the claimed. □

Pisinger [382] proposed to improve the above algorithm by also deleting in Step 6 all items with $w_{ij} \geq w_{ia_i}$ for each class N_i. The validity of this reduction follows from the fact that the current slope α should be increased, meaning that items with $w_{ij} > w_{ia_i}$ will not be considered. In Step 7, all items with $p_{ij} \leq p_{ib_i}$ can be deleted, since the current slope α needs to be decreased, and hence items with larger profits $p_{ij} > p_{ib_i}$ will enter the LP-solution.

11.2.2 Bounds from Lagrangian Relaxation

Two different bounds can be obtained from Lagrangian relaxation. If we relax the weight constraint with a multiplier $\lambda \geq 0$ we get $L_1(\text{MCKP},\lambda)$ defined as

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij} - \lambda \left(\sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} - c \right) \\ \text{subject to} \quad & \sum_{j \in N_i} x_{ij} = 1, \quad i = 1,\ldots,m, \\ & x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \quad j \in N_i. \end{aligned} \tag{11.14}$$

The objective function may be written as

$$z(L_1(\text{MCKP},\lambda)) = \lambda c + \sum_{i=1}^{m} \sum_{j \in N_i} \left(p_{ij} - \lambda w_{ij}\right) x_{ij} = \lambda c + \sum_{i=1}^{m} \max_{j \in N_i} \left(p_{ij} - \lambda w_{ij}\right).$$

Hence the bound may be derived in $O(n)$ time by choosing the item with the largest Lagrangian profit in each class. The corresponding solution vector is $x_{ij} = 1$ for $j = \arg\max_{j \in N_i}(p_{ij} - \lambda w_{ij})$ and 0 otherwise. The tightest bound is found by solving the Lagrangian dual problem

$$U_2 := \min_{\lambda \geq 0} z(L_1(\text{MCKP}, \lambda)). \tag{11.15}$$

Notice, that the constraints of (11.14) define the convex hull of the integer-solutions to (11.14) and hence a number of properties hold as described in Section 3.8. First of all it implies that the best choice of Lagrangian multiplier λ is found as the dual variable associated with the capacity constraint in C(MCKP). Moreover, the bound U_2 with the best choice of Lagrangian multiplier λ will correspond to the bound obtained by LP-relaxation, hence $U_1 = U_2$.

As the bound from Lagrangian relaxing the capacity constraint is not stronger than the bound from LP-relaxation, we refer to the $O(n)$ algorithm for the latter. Dyer, Riha and Walker [125], however, presented a specialized algorithm for deriving U_2, which is specially designed for reducing states in a dynamic programming algorithm. The algorithm may be seen as a Lagrangian relaxation variant of the MCKP-Greedy algorithm. Like in the MCKP-Greedy algorithm, the most expensive part is the preprocessing step (reduction of classes, sorting) which however only needs to be made once. Subsequent bounds can then be derived in sub-linear time by searching forward or backward from the split solution.

A different relaxation can be obtained by relaxing the multiple-choice constraints. In this case, by using multipliers $\lambda_1, \ldots, \lambda_m \in \mathbb{R}$, we get $L_2(\text{MCKP}, \lambda)$ given by

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij} - \sum_{i=1}^{m} \lambda_i \left(\sum_{j \in N_i} x_{ij} - 1 \right) \\ \text{subject to} \quad & \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq c, \\ & x_{ij} \in \{0,1\}, \quad i = 1, \ldots, m, \quad j \in N_i. \end{aligned} \tag{11.16}$$

Defining modified profit values $\tilde{p}_{ij} := p_{ij} - \lambda_i$ the objective function may be rewritten as

$$z(L_2(\text{MCKP}, \lambda)) := \sum_{i=1}^{m} \lambda_i + \sum_{i=1}^{m} \sum_{j \in N_i} (p_{ij} - \lambda_i) x_{ij} = \sum_{i=1}^{m} \lambda_i + \sum_{i=1}^{m} \sum_{j \in N_i} \tilde{p}_{ij} x_{ij}. \tag{11.17}$$

Thus the relaxation leads to a standard knapsack problem with profits $\tilde{p}_{ij}$. The resulting upper bound

$$U_3 := \min_{\lambda \in \mathbb{R}^m} z(L_2(\text{MCKP}, \lambda), \tag{11.18}$$

is obtained by choosing the multipliers $\lambda_1, \ldots, \lambda_m$ leading to the tightest bound.

Since the solution of $z(L_2(\text{MCKP},\lambda))$ involves the solution of a (KP), one may prefer to use any polynomial bounding procedure for (KP) as described in Section 5.1.1 to obtain polynomial upper bounds for (MCKP).

11.2.3 Other Bounds

Enumerative bounds (see Section 5.1.1) have been considered in Pisinger [382]. Some work on the polyhedral properties of (MCKP) have been presented by Johnson and Padberg [254] as well as Ferreira, Martin and Weismantel [143] — although only for the case where the multiple-choice constraint is in the weaker form $\sum_{j\in N_i} x_{ij} \leq 1$. See also Section 3.10 for a more detailed study of the knapsack polytope.

11.3 Class Reduction

Using the concept from Section 3.2 several decision variables may be fixed a-priori at their optimal value through class reduction. Let U_{ij}^1 be an upper bound on (MCKP) with the additional constraint $x_{ij} = 1$, and assume that z is the current incumbent solution value. If $U_{ij}^1 \leq z$ then we may fix x_{ij} at zero. Similarly, if U_{ij}^0 is an upper bound on (MCKP) with additional constraint $x_{ij} = 0$ and $U_{ij}^0 \leq z$, then we may fix x_{ij} at one, and hence all other decision variables in the class at zero. If the reduced set has only one item left, say item j, we fathom the class, fixing x_{ij} at 1.

Dyer, Kayal and Walker [124] presented a bound similar to the Dembo and Hammer bound [104] for the ordinary (KP) as presented in Section 5.1.3. Assume that the LP-relaxation $C(\text{MCKP})$ has been solved and that the fractional variables are $t_s, t_s' \in N_s$. en upper bound U_{ij}^1 is now determined by relaxing the constraint (11.3) for the fractional variables to $x_{st_s}, x_{st_s'} \in \mathbb{R}$. In this way, the bound U_{ij}^1 can be calculated as

$$U_{ij}^1 := \hat{p} - p_{it_i} + p_{ij} + \alpha^*(c - \hat{w} + w_{it_i} - w_{ij}). \tag{11.19}$$

Since each bound is calculated in constant time, the time complexity of reducing class N_i is $O(n_i)$.

Tighter, but more time-consuming bounds can be obtained by using the LP bound $U_1 = C(\text{MCKP})$ for the reduction. Since each bound is derived in $O(n)$ time, the complexity of reducing class N_i is $O(nn_i)$. Notice, that it is not worth testing with additional constraint $x_{ij} = 0$ as the corresponding bound U_{ij}^0 will lead to the trivial bound $U_{ij}^0 = C(\text{MCKP})$ which always is larger than z.

It may be beneficial to solve $C(\text{MCKP})$ through the MCKP-Greedy algorithm, as the time consuming reduction of classes in Step 1 only needs to be performed once. Moreover, in Step 4, one may start from the current split solution and just move a

few steps forward or backward to find the split solution for the problem with constraint $x_{ij} = 1$ imposed. Since empirically only a few iterations need to be performed in Step 4, the observed running time is constant for most instances considered.

Finally, Dudzinski and Walukiewicz [117] propose bounds obtained from the Lagrangian relaxation L_1(MCKP) in (11.14) obtaining an $O(n)$ running time for the reduction.

11.4 Branch-and-Bound

Several enumerative algorithms for (MCKP) have been presented during the last two decades: Nauss [355], Sinha and Zoltners [436], Dyer, Kayal and Walker [124], Dudzinski and Walukiewicz [117], Dyer, Riha and Walker [125]. In order to obtain an upper bound for the problem, most of these algorithms start by solving C(MCKP) in two steps:

1. The LP-dominated items are removed as described in Section 11.2.
2. The reduced C(MCKP) is solved by a greedy-type algorithm.

After these two initial steps, a lower bound z^ℓ is obtained by use of various heuristics which are going to be described in Section 11.9. Next, upper bound tests are used to fix several variables in each class to their optimal value. The reduced (MCKP) problem is then solved to optimality through enumeration.

To illustrate a particularly well-designed branch-and-bound algorithm, the algorithm by Dyer, Kayal and Walker is outlined in Figure 11.7. The algorithm makes use of the LP-based bounds C(MCKP) in the reduction phase as well as in the branch-and-bound phase.

Algorithm DyerKayalWalker:

1. Remove LP-dominated items as described in Section 11.2.
   Solve the LP-relaxation C(MCKP) to derive an upper bound.
2. Reduce the classes as described in Section 11.3.
3. Solve the remaining problem through branch-and-bound.

Fig. 11.7. The algorithm DyerKayalWalker is a classic three-phase branch-and-bound algorithm which solves the LP-relaxation and reduces the classes before solving the problem through branch-and-bound. Several other algorithms for the (MCKP) are based on the above framework.

In Step 3, the bounds are derived by solving C(MCKP) defined on the free variables. If $x_{st_s}, x_{st'_s}$ are the fractional variables of the LP solution, branching is performed by

first setting $x_{st'_s} = 1$ and then $x_{st'_s} = 0$. Backtracking is performed if either the upper bound $U = z(C(\text{MCKP}))$ is not larger than the current lower bound z^ℓ, or if the (MCKP) problem with the current variables fixed is infeasible. The branch-and-bound algorithm follows a depth-first search to limit the space consumption, and branching is always performed on the split items.

Dyer, Kayal and Walker furthermore improve the lower bound z^ℓ at every branching node by using the heuristic bound (11.28). For classes larger than $n_i \geq 25$ they also propose to use the class reduction from Section 11.3 at every branching node.

11.5 Dynamic Programming

(MCKP) can be solved in pseudopolynomial time through dynamic programming as shown by Dudzinski and Walukiewicz [117]. Let $z_\ell(d)$ be an optimal solution value to (MCKP) defined on the first ℓ classes and with restricted capacity d

$$z_\ell(d) := \max \left\{ \sum_{i=1}^{\ell} \sum_{j \in N_i} p_{ij} x_{ij} \;\middle|\; \begin{array}{l} \sum_{i=1}^{\ell} \sum_{j \in N_i} w_{ij} x_{ij} \leq d, \\ \sum_{j \in N_i} x_{ij} = 1,\ i = 1, \ldots, \ell, \\ x_{ij} \in \{0,1\},\ i = 1, \ldots, \ell,\ j \in N_i \end{array} \right\}, \tag{11.20}$$

where we assume that $z_\ell(d) := -\infty$ if no solution exists. Initially we set $z_0(d) := 0$ for all $d = 0, \ldots, c$, To compute $z_\ell(d)$ for $\ell = 1, \ldots, m$ we can use the recursion:

$$z_\ell(d) = \max \begin{cases} z_{\ell-1}(d - w_{\ell 1}) + p_{\ell 1} & \text{if } 0 \leq d - w_{\ell 1}\,, \\ z_{\ell-1}(d - w_{\ell 2}) + p_{\ell 2} & \text{if } 0 \leq d - w_{\ell 2}\,, \\ \vdots & \\ z_{\ell-1}(d - w_{\ell n_\ell}) + p_{\ell n_\ell} & \text{if } 0 \leq d - w_{\ell n_\ell}\,, \end{cases} \tag{11.21}$$

where we assume that the maximum operator returns $-\infty$ if we are maximizing over an empty set. An optimal solution to (MCKP) is found as $z = z_m(c)$ and we obtain $z = -\infty$ if assumption (11.4) is violated. The space bound of the dynamic programming algorithm is $O(mc)$, while each iteration of (11.21) takes n_i time, demanding totally $O(c \sum_{i=1}^{m} n_i) = O(nc)$ operations.

Reversing the roles of profits and weights we can also perform *dynamic programming by profits* as described in Section 2.3 for the classical knapsack problem. Let $y_i(q)$ denote the minimal weight of a solution of the subproblem (MCKP) consisting of the classes $N_1, \ldots, N_i$ with total profit equal to q. If no solution with profit value q exists, we will set $y_i(q) := c + 1$. A possible upper bound on the optimal solution value would be to use $U := 2z^h$ from (11.29) which is at most twice as large as the optimal solution value. Initializing $y_0(0) := 0$ and $y_0(q) := c + 1$ for $q = 1, \ldots, U$, the values for the classes $1, \ldots, m$ can be calculated for $i = 1, \ldots, m$ and $q = 0, \ldots, U$ by use of the recursion

$$y_i(q) = \min \begin{cases} y_{i-1}(q - p_{i1}) + w_{i1} & \text{if } 0 \le q - p_{i1}\,, \\ y_{i-1}(q - p_{i2}) + w_{i2} & \text{if } 0 \le q - p_{i2}\,, \\ \vdots & \\ y_{i-1}(q - p_{in_i}) + w_{in_i} & \text{if } 0 \le q - p_{in_i}\,, \end{cases} \tag{11.22}$$

The optimal solution is given by $\max\{q \mid y_m(q) \le c\}$. The total running time for all operations (11.22) is of order $O(U \sum_{i=1}^{m} n_i) = O(nU)$.

The recursion (11.21) has the drawback that an optimal solution is not reached before all classes have been enumerated, meaning that we have to pass through all $O(nc)$ steps. To avoid this problem, Pisinger [382] proposed a generalization of the primal-dual dynamic programming described in Section 5.3. Assume that the classes are reordered according to some global considerations, guessing that the last classes have a large probability for being fixed at their LP-optimal value.

Let $z_\ell(d)$, $d = 0,\ldots,2c$, be an optimal solution to the following (MCKP) problem defined on the first ℓ classes, where variables in classes after ℓ are fixed at their LP-optimal values:

$$z_\ell(d) := \max \left\{ \begin{array}{l} \sum_{i=1}^{\ell} \sum_{j \in N_i} p_{ij} x_{ij} \\ \quad + \sum_{i=\ell+1}^{m} p_{it_i} \end{array} \;\middle|\; \begin{array}{l} \sum_{i=1}^{\ell} \sum_{j \in N_i} w_{ij} x_{ij} + \sum_{i=\ell+1}^{m} w_{it_i} \le d, \\ \sum_{j \in N_i} x_{ij} = 1,\ i = 1,\ldots,\ell, \\ x_{ij} \in \{0,1\},\ i = 1,\ldots,\ell,\ j \in n_i \end{array} \right\}, \tag{11.23}$$

where items it_i for $i = 1,\ldots,m$ correspond to the split solution. Initially we set $z_0(d) := \hat{p}$ for all $d \ge \hat{w}$, and $z_0(d) := -\infty$ for all $d < \hat{w}$. Then the following recursion is applied:

$$z_\ell(d) = \max \begin{cases} z_{\ell-1}(d - w_{\ell 1} + w_{\ell t_\ell}) + p_{\ell 1} - p_{\ell t_\ell} & \text{if } 0 \le d - w_{\ell 1} + w_{\ell t_\ell} \le 2c, \\ z_{\ell-1}(d - w_{\ell 2} + w_{\ell t_\ell}) + p_{\ell 2} - p_{\ell t_\ell} & \text{if } 0 \le d - w_{\ell 2} + w_{\ell t_\ell} \le 2c, \\ \vdots & \\ z_{\ell-1}(d - w_{\ell n_\ell} + w_{\ell t_\ell}) + p_{\ell n_\ell} - p_{\ell t_\ell} & \text{if } 0 \le d - w_{\ell n_\ell} + w_{\ell t_\ell} \le 2c. \end{cases} \tag{11.24}$$

An optimal solution to (MCKP) is found as $z = z_m(c)$, obtaining $z = -\infty$ if assumption (11.4) is violated. We may think of the classes $C = \{1,\ldots,\ell\}$ as an *expanding core*.

The recursion (11.24) demands $O(n_i)$ operations for each class in the core and for each capacity d, yielding the complexity $O(\sum_{i=1}^{m} 2cn_i) = O(nc)$ for a complete enumeration. The space complexity is $O(2mc)$. However if optimality of a state can be proved after enumerating classes up to N_ℓ, then we may terminate the process, having used the computational effort $O(c \sum_{i=1}^{\ell} n_i)$.

As for (KP) it is convenient also for (MCKP) to run dynamic programming with lists (analogous to algorithm DP-with-Lists), as described in Section 3.4, having each list represented by the pair $(\overline{w}_i, \overline{p}_i)$, where $\overline{p}_i = z_\ell(\overline{w}_i)$.

The separability property for (KP) as presented by Horowitz and Sahni [238] is easily generalized to (MCKP). The classes should be separated in two sets such that $n_1 \cdot n_2 \cdots n_\ell$ and $n_{\ell+1} \cdot n_{\ell+2} \cdots n_m$ are of same magnitude and then use normal dynamic programming on each of the two problems. As for (KP), the states are easily merged in linear time. This leads to a time bound of $O(\min\{nc,\ n_1 \cdots n_\ell,\ n_{\ell+1} \cdots n_m\})$.

Using word-parallelism Pisinger [392] presented a dynamic programming algorithm which runs in $O(nM/\log M)$ time and $O(n + M/\log M)$ space, where $M = \max\{c, z^*\}$. The algorithm is a straightforward adaption of the Word RAM algorithm for (KP) presented in Section 5.2.1: Two tables g and h are maintained with bitmaps corresponding to the undominated states. For each class $i = 1, \ldots, m$, we obtain one result table by applying the Wordmerge algorithm n_i times corresponding to the items in class N_i.

Finally, one could try to use the concept of balancing as presented in Section 3.6. However, no balanced algorithm for (MCKP) in the general form has been published, but the concept has been applied to the multiple-choice subset sum problem which we will consider in Section 11.10.1.

11.6 Reduction of States

During a branch-and-bound algorithm it is necessary to derive upper bounds for the considered branching nodes or states. Upper bounds are also crucial when running dynamic programming with upper bounds as described in Section 3.5. The bounds could be derived in $O(n)$ using the algorithm Dyer-Zemel, but specialized methods yield a better performance.

Dudzinski and Walukiewicz [115] derived an efficient bounding technique as follows: Initially the classes are reduced according to (11.6) and (11.7) obtaining classes R_i. Then, each time an upper bound should be derived, a median search algorithm is used in the sorted classes R_i, finding the LP bound in $O(m \log^2(n'/m))$ where n' is the number of undominated items.

Weaker bounds, which are however calculated in constant time once some preprocessing has been done, were proposed in Pisinger [382]. In order to derive these bounds we first need to define positive and negative gradients α_i^+ and α_i^- for each class N_i, $i \neq s$. The gradients are defined as (see Figure 11.8):

$$\begin{aligned} \alpha_i^+ &:= \max_{j \in N_i,\ w_{ij} > w_{it_i}} \frac{p_{ij} - p_{it_i}}{w_{ij} - w_{it_i}}, \quad i = 1, \ldots, m,\ i \neq s, \\ \alpha_i^- &:= \min_{j \in N_i,\ w_{ij} < w_{it_i}} \frac{p_{it_i} - p_{ij}}{w_{it_i} - w_{ij}}, \quad i = 1, \ldots, m,\ i \neq s, \end{aligned} \tag{11.25}$$

where we set $\alpha_i^+ = 0$ (respectively $\alpha_i^- = \infty$) if the set we are maximizing (respectively minimizing) over is empty. Note that α_i^+ and α_i^- can be derived in $O(n_i)$ for

each class N_i and they do not demand any preprocessing. The gradients are a measure of the expected gain (respectively loss) per weight unit by choosing a heavier (respectively lighter) item from N_i instead of the LP-optimal choice t_i.

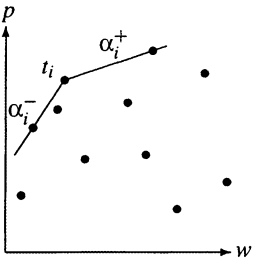


Fig. 11.8. Gradients α_i^+, α_i^- in class N_i.

Assume that an *expanding core* is enumerated using recursion (11.24) up to class N_ℓ. Moreover define for each class i the extreme gradients among the successive classes $\ell > i$ as

$$\hat{\alpha}_i^+ := \max_{\ell > i} \alpha_\ell^+, \qquad \hat{\alpha}_i^- := \min_{\ell > i} \alpha_\ell^-. \tag{11.26}$$

Then the bound on a state with profit $\overline{p}$ and weight $\overline{w}$ may be found as

$$U(\overline{w}, \overline{p}) := \begin{cases} \overline{p} + (c - \overline{w})\hat{\alpha}_\ell^+ & \text{if } \overline{w} \le c, \\ \overline{p} + (c - \overline{w})\hat{\alpha}_\ell^- & \text{if } \overline{w} > c. \end{cases} \tag{11.27}$$

Notice that the best bounds are obtained by ordering the classes such that the classes with α_i^+, α_i^- closest to α^* are placed first.

11.7 Hybrid Algorithms and Expanding Core Algorithms

Dyer, Riha and Walker [125] presented a hybrid algorithm based on the concept of dynamic programming with upper bound tests as described in Section 3.5.

The algorithm starts by removing dominated items using test (11.6). Then it runs the dynamic programming recursion (11.21) using dynamic programming with lists, DP-with-Lists, as described in Section 3.4. For each iteration of recursion (11.21), i.e. for each class added, upper bound tests are used to fathom states which will not lead to an optimal solution. The bound used is U_2 based on Lagrangian relaxation of the capacity constraint described in Section 11.2.2. An outline of the algorithm is found in Figure 11.9.

As for other knapsack problems it may be beneficial to focus the enumeration on a *core* when dealing with large-sized problems. Basically one should distinguish

Algorithm DyerRihaWalker:
for $i := 1$ to m
    derive class R_i from N_i
    use recursion (11.21) to obtain states $(\overline{w}, \overline{p}) := (\overline{w}, z_i(\overline{w}))$
    remove infeasible and dominated states $(\overline{w}, \overline{p})$
    for each remaining state $(\overline{w}, \overline{p})$
        derive an upper bound $U(\overline{w}, \overline{p})$ and a lower bound $z^\ell(\overline{w}, \overline{p})$ by solving the Lagrangian relaxation (11.14) and rounding it down
    let $z^\ell := \max z^\ell(\overline{w}, \overline{p})$
    let $U := \min U(\overline{w}, \overline{p})$
    if $z^\ell = U$ then stop
end for

Fig. 11.9. The algorithm DyerRihaWalker is a hybrid algorithm combining branch-and-bound with dynamic programming.

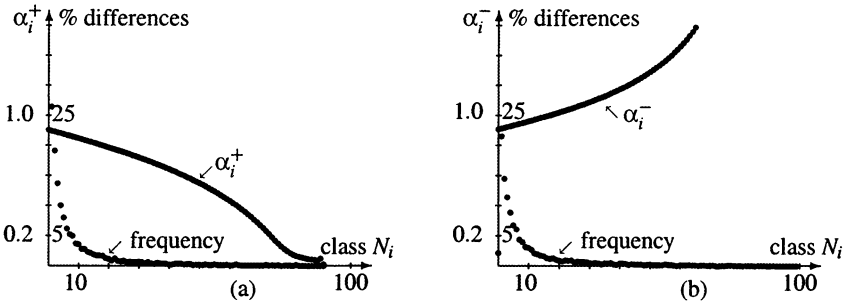


Fig. 11.10. Frequency of classes N_i where the integer optimal choice differs from LP-optimal choice, compared to gradient α_i^+ respectively α_i^-.

between two kinds of a core when dealing with (MCKP): A core where only a subset $C \subset K = \{1, \ldots, m\}$ of the classes is enumerated, or a core where only some items $j \in C_i \subset N_i$ in each class are enumerated. The first approach is suitable for problems with many classes, while the second approach is more appropriate for problems with many items in each class. No work has however been published on the second approach, thus we will here consider the case where a core consists of a subset of the classes.

Pisinger [382] defined a core problem based on positive and negative gradients α_i^+ and α_i^- for each class N_i, $i \neq s$. The concept is motivated by the following observation:

In Figure 11.10 (a) we have ordered the classes according to decreasing α_i^+ and show how often the integer optimal solution to (MCKP) differs from the LP-optimal choice in each class N_i. The figure is a result of 5000 randomly generated instances ($m = 100$, $n_i = 10$), where we have measured how often the integer optimal choice j (satisfying $w_{ij} > w_{it_i}$ since we are considering forward gradients) differs from the

LP-optimal choice t_i in each class N_i. It can be seen that when α_i^+ decreases, so does the probability that t_i is not the integer optimal choice. Similarly, in Figure 11.10 (b) we have ordered the classes according to increasing α_i^- to show how the probability for changes decreases with increased α_i^-.

This observation motivates the consideration of only a small number of the classes N_i, namely those classes where α_i^+ or α_i^- are sufficiently close to α^* given by (11.9) to (11.11). Thus a fixed-core algorithm for (MCKP) can be derived by straightforward generalization of (KP) algorithms: Choose a subset $C \subset K$ of the classes where the gradients are close to α. These classes may be obtained in $O(m)$ by choosing the δ classes with the largest value of α_i^+. In a similar way additional δ classes with smallest value of α_i^- are chosen. This gives a core C of *at most* 2δ classes, since some classes may have been chosen twice, and thus should be eliminated.

Following the concept of Section 5.4 a fixed-core algorithm should enumerate the classes in the core, and then try to fix the remaining classes at their optimal values through the reduction rules described in Section 11.3. If all items in classes $N_i \notin C$ can be fixed at their LP-optimal values, the solution to the core problem is optimal. Otherwise the reduced classes must be enumerated to completion either by branch-and-bound or dynamic programming. No fixed-core algorithm has presently been published for the (MCKP).

An expanding core algorithm for (MCKP) named Mcknap has been published by Pisinger [382]: Initially the algorithm solves the LP problem, using a simplified version of Dyer-Zemel. Then the classes are ordered such that classes N_i with gradient α_i^+ or α_i^- closest to α^* are first. Initially only the split class N_s is enumerated, while other classes are introduced consecutively in recursion (11.24). Upper bound tests are used to fathom states which cannot lead to an optimal solution, hoping that optimality of the problem can be proved with classes $N_i \notin C$ fixed at their LP-optimal value. Before a class is added to the core, it is reduced by using the tests (11.19), and if some items cannot be fixed at their LP-optimal values, these are sorted and then reduced according to (11.6) and (11.7). The algorithm is illustrated in Figure 11.11.

The procedure add(L,R) with $R = \{(p_1,w_1),\ldots,(p_r,w_r)\}$ merges the r lists $(L \oplus (p_1,w_1)),\ldots,(L \oplus (p_r,w_r))$ using the Merge-Lists algorithm presented in Section 3.4.

The procedure reduceset(L) reduces the list of states L as described in Section 3.5 using the bound (11.27). The procedure also finds an incumbent solution as $z' := \max_{(\overline{p}_i,\overline{w}_i)\in L}\{\overline{p}_i | \overline{w}_i \le c\}$. If z' exceeds the present best solution z^ℓ we update z^ℓ accordingly.

Finally the procedure reduceclass(N) first applies the upper bound test (11.19) to fix some variables at their optimal values. This part of the algorithm can be performed in $O(n_i)$ time. Then it deletes dominated items from the class N using criteria (11.6) in time $O(n_i \log n_i)$.

In order to speed up the solution process for instances with many classes, the sorting of classes according to gradients A^+ and A^- and the reduction of classes according

Algorithm McKnap:

Solve C(MCKP) by a randomized version of the Dyer-Zemel algorithm.
Determine gradients $A^+ = \{\alpha_i^+\}$ and $A^- = \{\alpha_i^-\}$ for $i = 1,\ldots,m,\ i \neq s$.
Partially sort A^+ in decreasing order and A^- in increasing order.
$z^\ell := 0$, $a := 1$, $b := 1$, $C := \{N_s\}$, $L_C :=$ reduceclass(N_s)
repeat
    reduceset(L_C)
    if $L_C = \emptyset$ then stop
    $i :=$ index(A_a^-), $a := a+1$ *choose next class from* A^-
    if N_i has not been considered then
        $R_i :=$ reduceclass(N_i)
        if $|R_i| > 1$ then add(L_C, R_i)
    end if
    reduceset(L_C)
    if $L_C = \emptyset$ then stop
    $i :=$ index(A_b^+), $b := b+1$ *choose next class from* A^+
    if N_i has not been considered then
        $R_i :=$ reduceclass(N_i)
        if $|R_i| > 1$ then add(L_C, R_i)
    end if
forever
Find the solution vector.

Fig. 11.11. The Mcknap algorithm is an expanding-core algorithm based on dynamic programming with lists.

to (11.19) is performed in a lazy way using the same concept as in the Minknap algorithm (Section 5.4.2). Since each class appears in both sets A^+,A^- we have to test whether the class N_i already have been considered in the expanding core algorithm.

11.8 Computational Experiments

In this section we will experimentally compare the performance of algorithms DyerKayalWalker, DyerRihaWalker and Mcknap, in order to see the effect of using dynamic programming as opposed to branch-and-bound, and of using an expanding core as opposed to a complete enumeration.

Five types of randomly generated instances are considered, each instance tested with data-range R chosen as $R_1 = 1000$ or $R_2 = 10000$ for different numbers of classes m and sizes n_i as follows.

- *Uncorrelated instances*: In each class we generate n_i items by choosing w_{ij} and p_{ij} uniformly in $[1,R]$.

		Uncorrelated		Weakly corr.		Strongly corr.		Subset sum		Monotone	
m	n_i	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2
10	10	0	0	6	50	447	649	2	9	0	1
100	10	1	1	11	—	—	—	1	21	4	6
1000	10	9	21	8	—	—	—	3	19	12	113
10000	10	76	307	81	165	—	—	35	35	94	585
10	100	0	0	5	—	—	—	3	34	0	1
100	100	3	1	16	97	—	—	3	220	8	13
1000	100	55	159	69	1804	—	—	33	206	80	256
10	1000	7	4	8	—	—	—	9	199	7	34
100	1000	47	47	61	784	2176	—	49	49	51	66

Table 11.1. Total computing time, DyerKayalWalker, in milliseconds (INTEL PENTIUM 4, 1.5 GHz). Averages of 100 instances.

- *Weakly correlated instances*: In each class, w_{ij} is uniformly distributed in $[1,R]$ and p_{ij} is uniformly distributed in $[w_{ij}-10, w_{ij}+10]$, such that $p_{ij} \geq 1$.
- *Strongly correlated instances*: For (KP) these instances are generated with w_j uniformly distributed in $[1,R]$ and $p_j = w_j + 10$, which are difficult instances for (KP). These instances are however trivial for (MCKP), since they degenerate to subset sum instances (SSP), but hard instances for (MCKP) may be constructed by cumulating strongly correlated instances for (KP): For each class generate n_i items (p'_j, w'_j) as for (KP), and order these by increasing weights. The instance for (MCKP) is then $w_{ij} = \sum_{h=1}^{j} w'_h$, $p_{ij} = \sum_{h=1}^{j} p'_h$, $j = 1,\ldots,n_i$. Such instances have no dominated items, and form an upper convex hull.
- *Subset-sum instances*: w_{ij} uniformly distributed in $[1,R]$ and $p_{ij} = w_{ij}$.
- *Monotone instances*: Sinha and Zoltners [436] presented some instances with very few dominated items. For each class construct n_i items as (p'_j, w'_j) uniformly distributed in $[1,R]$. Order the profits in increasing order and set $p_{ij} := p'_j$ for $j = 1,\ldots,n_i$. Similarly order the weights in increasing order, and set $w_{ij} := w'_j$ for $j = 1,\ldots,n_i$. These instances are also denoted the *zig-zag instances* due to their appearance when plotted in the Euclidean space.

The codes DyerKayalWalker and DyerRihaWalker were obtained by courtesy from the authors. The code DyerRihaWalker has a number of parameters which can be used to tune the algorithm. The following tests were run using the default parameters.

The computational times obtained are given in Tables 11.1 to 11.3. The experiments were run on a INTEL PENTIUM 4, 1.5 GHz. A time limit of one hour was given to all the instances, and a dash in the tables indicates that not all instances could be solved within this limit.

The oldest of the codes DyerKayalWalker, which is based on a pure branch-and-bound framework, has good solution times for the uncorrelated, subset sum and monotone instances. However, when it comes to weakly correlated and strongly

		Uncorrelated		Weakly corr.		Strongly corr.		Subset sum		Monotone	
m	n_i	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2
10	10	0	0	2	16	4	32	2	21	0	0
100	10	1	1	8	144	337	3214	3	26	4	5
1000	10	14	18	48	190	11321	—	18	41	61	96
10000	10	612	878	1082	1766	—	—	204	281	1863	2520
10	100	0	0	6	111	—	—	6	171	5	7
100	100	5	5	16	133	—	—	15	177	43	110
1000	100	54	63	174	391	—	—	114	336	655	1125
10	1000	4	4	14	301	1059	—	57	1020	30	229
100	1000	46	48	82	647	14931	—	134	1139	224	2547

Table 11.2. Total computing time, DyerRihaWalker, in milliseconds (INTEL PENTIUM 4, 1.5 GHz). Averages of 100 instances.

		Uncorrelated		Weakly corr.		Strongly corr.		Subset sum		Monotone	
m	n_i	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2
10	10	0	0	1	9	2	14	1	20	0	0
100	10	0	0	2	31	61	561	1	14	1	1
1000	10	3	3	4	30	964	9432	1	12	5	8
10000	10	26	34	28	44	18440	152166	15	14	38	48
10	100	0	0	6	83	4	29	10	134	2	3
100	100	1	1	5	81	52	828	1	89	6	10
1000	100	11	12	22	64	1419	19484	10	11	32	40
10	1000	2	2	19	378	335	20	2	1138	33	136
100	1000	8	10	25	155	15835	458	12	15	60	411

Table 11.3. Total computing time, Mcknap, in milliseconds (INTEL PENTIUM 4, 1.5 GHz). Averages of 100 instances.

correlated instances, several instances cannot be solved to optimality within the time limit.

The two codes based on dynamic programming, DyerRihaWalker and Mcknap have an overall stable behavior, where Mcknap has the fastest solution times. There are two reasons for the faster solution times of Mcknap: For easy instances, Mcknap takes benefit of the expanding core algorithm so that it can derive bounds in linear time, and does not need to consider all items. For difficult instances, demanding a considerable enumeration in the dynamic programming part, the Mcknap algorithm takes advantage of constant time bounds for the reduction.

It is however interesting to see, that the Mcknap algorithm does not perform so well for instances with a few very large classes. This can be explained by the fact, that the core is defined as a subset of classes. If the classes are large, the core concept is not able to focus the search on the relevant items. Other definitions of a core could however be applied in these situations as discussed in Section 11.7.

11.9 Heuristics and Approximation Algorithms

The split solution $\hat{x}$, taking the LP-optimal choices x_{it_i} in each class (see Section 11.2) is generally a good heuristic solution of value $z' = \hat{p}$. The relative performance of the heuristic is however arbitrarily bad, as can be seen with the instance $m := 2$, $n_1 := n_2 := 2$, $c := M$ and items $(p_{11}, w_{11}) := (0,0)$, $(p_{12}, w_{12}) := (M, M)$, $(p_{21}, w_{21}) := (0,0)$ and $(p_{22}, w_{22}) := (2,1)$. The split solution is given by items 1,1 and 2,2 yielding $z' = \hat{p} = 2$ although the optimal objective value is $z^* = M$.

Dyer, Kayal and Walker [124] presented an improvement algorithm for (MCKP) which runs in $O(\sum_{i=1}^{m} n_i) = O(n)$ time. Let $\beta_i = t_i$ for classes $i \neq s$ and $\beta_s = t'_s$, where t'_s is one of the fractional variables found by Algorithm MCKP-Greedy. Thus, by setting $x_{i\beta_i} = 1$ in each class, we obtain an infeasible solution. Now, iteratively for each class $i = 1, \ldots, m$, find the item ℓ_i which when replacing item β_i, gives a feasible solution with the largest profit. Thus $\ell_i := \arg\max_{j \in N_i} \{p_{ij} | w_{ij} - w_{i\beta_i} + \hat{w} \leq c\}$ and let

$$z^r := \max_{i=1,\ldots,m} \{\hat{p} + p_{i\ell_i} - p_{i\beta_i}\}. \tag{11.28}$$

This heuristic however also has an arbitrarily bad performance ratio which can be seen with the instance $N_1 := N_2 := N_3 := \{(0,0), (1,1), (M, M+1)\}$ and $m := 3$, $c := M + 1$. We have the heuristic value $z' = \hat{p} = 3$, and also $z^r = 3$ although $z^* = d$.

It is possible to obtain a heuristic solution with worst-case performance $\frac{1}{2}$ by setting

$$z^h := \max\{z', z^s\}, \tag{11.29}$$

where $z^s := p_{st'_s} + \sum_{i \neq s} p_{i,\alpha_i}$ with $\alpha_i = \arg\min_{j \in N_i} \{w_{ij}\}$. In other words z^s is the sum of the split item from the split class, and the sum of profits of the lightest item in each of the other classes. Notice that z^s is a feasible solution according to (11.5). Obviously $z^* \leq z' + z^s$, thus $z^* \leq 2z^h$. To see that the bound is tight, consider the (KP) instance given in the proof of Theorem 2.5.4, transformed to a (MCKP). We have $m = 3$ classes of size $n_i = 2$. The items are $p_{11} = 2$, $w_{11} = 1$, $p_{21} = w_{21} = p_{31} = w_{31} = M$ and $p_{i2} = w_{i2} = 0$ for $i = 1, \ldots, 3$. The capacity is $c = 2M$. Assuming that the LP-optimal solution chooses items (11), (21) and a fraction of item (31), we get $z^h = M + 1$ where the optimal solution is $z^* = 2M$. Hence z^h / z^* is arbitrarily close to $\frac{1}{2}$ for sufficiently large values of M.

An approximation algorithm for (MCKP) with performance guarantee 4/5 was given by Gens and Levner [171]. Their algorithm is based on binary search and runs in $O(n \log m)$ time.

For developing an *FPTAS* for (MCKP) we use *dynamic programming by profits* as given by recursion (11.22). This recursion can be adapted to an *FPTAS* analogously to Section 2.6 for (KP). The approach relies again on appropriate scaling of the profit values and running recursion (11.22) with the scaled profit values $\tilde{p}_{ij} := \frac{p_{ij}}{K}$ with an appropriately chosen constant K. In order to get a performance guarantee of $1 - \varepsilon$, constant K must be chosen analogously to (2.19), namely

$$K \leq \frac{\varepsilon z^*}{m}. \tag{11.30}$$

The choice of $K := \frac{2\varepsilon z^h}{m}$ clearly satisfies condition (11.30). Since

$$\frac{U}{K} = \frac{Um}{2\varepsilon z^h} \leq \frac{m}{\varepsilon},$$

for $U \leq 2z^h$, we can replace U in recursion (11.22) by $\frac{m}{\varepsilon}$. Taking into account that U can be calculated in linear time, we get an overall running time of $O(\frac{nm}{\varepsilon})$. Note that because of the multiple-choice structure a further separation of the items into "small" and "large" items like in Chapter 6 for (KP) does not seem to be meaningful.

The first fully polynomial time approximation scheme for (MCKP) has been given by Chandra, Hirschberg and Wong [77]. The above presented *FPTAS* is similar to the *FPTAS* by Lawler [295]. Its running time is significantly better than the running time of the scheme in [77].

11.10 Variants of the Multiple-Choice Knapsack Problem

11.10.1 Multiple-Choice Subset Sum Problem

If $p_{ij} = w_{ij}$ in all classes N_i, $i = 1,\ldots,m$, then the problem may be seen as a *multiple-choice subset sum problem* (MCSSP). In this problem we have m classes, each class N_i containing weights $w_{i1},\ldots,w_{in_i}$. The problem is to select exactly one weight from each class such that the total weight sum is maximized without exceeding the capacity c. Thus we have

$$\begin{aligned}
\text{(MCSSP)} \qquad & \text{maximize} \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \\
& \text{subject to} \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq c, \\
& \sum_{j \in N_i} x_{ij} = 1, \quad i = 1,\ldots,m, \\
& x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \quad j \in N_i,
\end{aligned} \tag{11.31}$$

where $x_{ij} = 1$ if item j was chosen in class N_i.

Pisinger [387] presented an algorithm for (MCSSP) based on balancing as described in Section 4.1.5. Let $\alpha_i := \arg\min_{j \in N_i}\{w_{ij}\}$ and $\beta_i := \arg\max_{j \in N_i}\{w_{ij}\}$ for $i = 1,\ldots,m$ be the indices of the smallest and largest weight in each class i. The

Algorithm Balmcsub:

```
1  for w̄ := c − w_max + 1 to c do g(s − 1, w̄) := 0
2  for w̄ := c + 1 to c + w_max do g(s − 1, w̄) := 1
3  g(s − 1, ŵ) := s
4  for b := s to m do
5     for w̄ := c − w_max + 1 to c + w_max do g(b, w̄) := g(b − 1, w̄)
6     for w̄ := c − w_max + 1 to c do
7        for i ∈ N_b do
8           w̄' := w̄ + w_bi − w_bα_b, g(b, w̄') := max{g(b, w̄'), g(b − 1, w̄)}
9     for w̄ := c + w_b down to c + 1 do
10       for j := g(b − 1, w̄) to g(b, w̄) − 1 do
11          for i ∈ N_j do
12             w̄' := w̄ + w_ji − w_jβ_j, g(b, w̄') := max{g(b, w̄'), j}
```

Fig. 11.12. Algorithm Balmcsub is a generalization of the Balsub algorithm based on balancing as described in Section 4.1.5.

split class s is defined by $s = \min\{j | \sum_{i=1}^{j} w_{i\beta_i} > c - \sum_{i=j+1}^{m} w_{i\alpha_i}\}$. The value of the *split solution* is $\hat{w} = \sum_{i=1}^{s-1} w_{i\beta_i} + \sum_{i=s}^{m} w_{i\alpha_i}$. With these definitions, the balanced algorithm can be described as in Figure 11.12.

Algorithm Balmcsub follows the same approach as algorithm Balsub described in Section 4.1.5, hence we refer to the latter for a detailed explanation.

A surprising result is that the algorithm runs in $O(w_{\max} \sum_{i=1}^{k} n_i) = O(nw_{\max})$ time and space, hence having the same time complexity as an ordinary (SSP).

Using word-parallelism, Pisinger [392] showed, that a dynamic programming algorithm may solve the problem in time of $O(n + c/\log c)$ by using the algorithm described at the end of Section 11.5. Instead of using Wordmerge we may use binary operations directly as shown in Section 4.1.1. The space complexity is also reduced to $O(n + c/\log c)$. The time and space complexity matches the best complexity for the ordinary (SSP).

11.10.2 Generalized Multiple-Choice Knapsack Problem

A different generalization of (MCKP) presented by Pisinger [389] is the *generalized multiple-choice knapsack problem* (GMCKP) in which we do not only demand a single item to be chosen from each class N_i, but any minimum-, maximum- or strict cardinality constraint can be assigned to a class. Formally the problem may be formulated as follows

$$\text{(GMCKP)} \qquad \text{maximize} \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij}$$

$$\begin{aligned}
\text{subject to } & \sum_{i=1}^{m}\sum_{j\in N_i} w_{ij}x_{ij} \le c, \\
& \sum_{j\in N_i} x_{ij} \le a_i, \quad i \in L, \\
& \sum_{j\in N_i} x_{ij} \ge a_i, \quad i \in G, \\
& \sum_{j\in N_i} x_{ij} = a_i, \quad i \in E, \\
& x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \quad j \in N_i.
\end{aligned} \tag{11.32}$$

The sets L, G, E form a partitioning of $\{1,\ldots,m\}$. All coefficients p_{ij}, w_{ij}, a_i and c are nonnegative integers, and the classes $N_1,\ldots,N_m$ are mutually disjoint.

If we relax the integrality constraint $x_{ij} \in \{0,1\}$ in (11.32) to $0 \le x_{ij} \le 1$ we obtain the *linear generalized multiple choice knapsack problem* C(GMCKP).

The interesting property of (GMCKP) is that it contains all the *weakly $\mathcal{NP}$-hard* knapsack problems as a special case, including the *subset sum problem*, *bounded knapsack problem*, and *multiple-choice knapsack problem*.

Every (GMCKP) can in polynomial time be put in a form with equalities in all the cardinality constraints as follows:

$$\begin{aligned}
\text{maximize } & \sum_{i=1}^{m}\sum_{j\in N_i} p_{ij}x_{ij} \\
\text{subject to } & \sum_{i=1}^{m}\sum_{j\in N_i} w_{ij}x_{ij} \le c, \\
& \sum_{j\in N_i} x_{ij} = a_i, \quad i = 1,\ldots,m, \\
& x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \quad j \in N_i.
\end{aligned} \tag{11.33}$$

The transformation is a stepwise modification of the constraints and item profits and weights:

Step 1

A class N_i with a constraint $\sum_{j\in N_i} x_{ij} \ge a_i$ is modified to a constraint of the form $\sum_{j\in N_i} x_{ij} \le n - a_i$ by adding $\sum_{j\in N_i} p_{ij}$ to the objective function and subtracting $\sum_{j\in N_i} w_{ij}$ from the capacity c. In addition, all items $j \in N_i$ change sign to $(-p_{ij}, -w_{ij})$.

Step 2

A class N_i with a constraint $\sum_{j\in N_i} x_{ij} \le a_i$ is easily modified to equality constraint $\sum_{j\in N_i} x_{ij} = a_i$ by adding a_i new items to the class which have profit and weight equal to zero.

Step 3

Negative profits and weights are handled by adding a sufficiently large constant to all items in a class N_i.

Pisinger [389] presented an algorithm for the optimal solution of (GMCKP) in the standard form (11.33) using *Dantzig-Wolfe decomposition*. The algorithm is based on the transformation to an equivalent (MCKP), which is achieved by defining for each class N_i a set of new items $(\overline{p}_{ik}, \overline{w}_{ik})$ for $k = 0, \ldots, c$ given by

$$\begin{aligned} \overline{w}_{ik} &:= k, \\ \overline{p}_{ik} &:= \max \left\{ \sum_{j=1}^{n_i} p_{ij} x_{ij} \;\middle|\; \begin{array}{l} \sum_{j=1}^{n_i} w_{ij} x_{ij} = k, \; \sum_{j=1}^{n_i} x_{ij} = a_i, \\ x_{ij} \in \{0,1\}, \; j = 1, \ldots, n_i \end{array} \right\}. \end{aligned} \tag{11.34}$$

The transformed profits $\overline{p}_{ik}$ for $k = 0, \ldots, c$ can be determined through dynamic programming using some of the recursions described in Section 2.3. The transformed class $\overline{N}_i$ has size $\overline{n}_i = c$, and the dynamic programming runs in $O(n_i a_i c)$ for each class of (GMCKP), using totally $O(c \sum_{i=1}^{m} n_i a_i)$ time to reach the transformed problem.

In this way we obtain a (MCKP) defined in classes $\overline{N}_i$, where the problem is to maximize the profit sum by choosing exactly one item from each class without exceeding the capacity constraint:

$$\begin{aligned} \text{maximize } & z = \sum_{i=1}^{m} \sum_{j \in \overline{N}_i} \overline{p}_{ij} \overline{x}_{ij} \\ \text{subject to } & \sum_{i=1}^{m} \sum_{j \in \overline{N}_i} \overline{w}_{ij} \overline{x}_{ij} \leq c, \\ & \sum_{j \in \overline{N}_i} \overline{x}_{ij} = 1, \quad i = 1, \ldots, m, \\ & \overline{x}_{ij} \in \{0,1\}, \quad i = 1, \ldots, m, \;\; j \in \overline{N}_i. \end{aligned} \tag{11.35}$$

There are c items in each class of (11.35), so the (MCKP) can be solved in $O(mc^2)$ through dynamic programming as seen in Section 11.5. This gives a total solution time for (GMCKP) of $O(c \sum_{i=1}^{m} n_i a_i + mc^2)$.

The LP-relaxation of (11.35) is solved through *column generation* (see also Section 15.2). In order to obtain an integer solution, an *expanding core approach* is used, where the classes gradually are transformed to the new form (11.35). Computational experiments show that only a very few classes need to be transformed, and hence the worst-case complexity of $O(mc^2)$ is very pessimistic.

11.10.3 The Knapsack Sharing Problem

Brown [56] introduced the *knapsack sharing problem* which in the binary form may be described as follows: Given a set $N = \{1, \ldots, n\}$ of items, each item $j \in N$ be-

longing to exactly one of the m disjoint classes $N_1, \ldots, N_m$, with $\cup_{i=1}^{m} N_i = N$. Item $j \in N_i$ has an associated profit p_{ij} and weight w_{ij}. The objective is to pack a subset of the items into a knapsack of capacity c such that the minimum of the profit sums in the classes is maximized. Using binary variables x_{ij} to denote whether item j was chosen in class N_i, we may formulate the knapsack sharing problem as:

$$\begin{aligned} \text{(KSP)} \qquad & \text{maximize} \min_{i=1,\ldots,m} \left\{ \sum_{j \in N_i} p_{ij} x_{ij} \right\} \\ & \text{subject to} \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq c, \\ & \qquad x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \; j \in N_i. \end{aligned} \tag{11.36}$$

The (KSP) is $\mathcal{NP}$-hard, as it reduces to (KP) for $m = 1$. The problem reflects situations where items belong to a number of owners, and each owner wishes to maximize the total profit of his own items in the shared knapsack. The model maximizes the minimum level of the profit sum that can be guaranteed to all owners. Yamada, Futakawa and Kataoka [491] describe a concrete application to a budgeting problem covering m districts of a given city. The set N_i is the set of projects related to the ith district. The city council wishes to select a set of projects such that each district gets a reasonable benefit of the investment.

Yamada, Futakawa and Kataoka [491] presented a dynamic programming algorithm for the solution of (KSP) having exponential running time. A better dynamic programming approach can be obtained by using a similar transformation technique as presented above for the (GMCKP).

For each class N_i we define a set of new items $(\overline{p}_{id}, \overline{w}_{id})$ for $d = 0, \ldots, c$ given by

$$\begin{aligned} & \overline{w}_{id} := d, \\ & \overline{p}_{id} := \max \left\{ \sum_{j=1}^{n_i} p_{ij} x_{ij} \;\middle|\; \begin{array}{l} \sum_{j=1}^{n_i} w_{ij} x_{ij} \leq d, \\ x_{ij} \in \{0,1\}, \; j = 1,\ldots,n_i \end{array} \right\}. \end{aligned} \tag{11.37}$$

The transformed profits $\overline{p}_{id}$ for $d = 0, \ldots, c$ can be determined through dynamic programming. The transformed class $\overline{N}_i$ has size $\overline{n}_i = c$, and the dynamic programming runs in $O(n_i c)$ for each class, using totally $O(nc)$ time.

In this way we obtain a max-min version of the (MCKP) given by

$$\begin{aligned} \text{(MaxMinMCKP)} \qquad & \text{maximize} \min_{i=1,\ldots,m} \sum_{j \in \overline{N}_i} \overline{p}_{ij} x_{ij} \\ & \text{subject to} \sum_{i=1}^{m} \sum_{j \in \overline{N}_i} \overline{w}_{ij} x_{ij} \leq c, \\ & \qquad \sum_{j \in \overline{N}_i} x_{ij} = 1, \quad i = 1,\ldots,m, \\ & \qquad x_{ij} \in \{0,1\}, \quad i = 1,\ldots,m, \; j \in \overline{N}_i. \end{aligned} \tag{11.38}$$

The latter problem can be solved efficiently through the greedy algorithm named MaxMinGreedy outlined in Figure 11.13.

Algorithm MaxMinGreedy:

1. Assume that the items in each class $\overline{N}_i$ are sorted such that $\overline{p}_{ij}$ and $\overline{w}_{ij}$ are increasing (possibly by removing dominated items).
2. Set $d_i := 1$ for $i = 1, \ldots, m$.
3. repeat
   choose $h := \arg\min_{i=1,\ldots,m} \overline{p}_{id_i}$
   if $\sum_{i=1}^{m} \overline{w}_{id_i} > c + \overline{w}_{hd_h} - \overline{w}_{h,d_h+1}$ then stop
   set $d_h := d_h + 1$
4. Return the solution $x_{id_i} := 1$ for $i = 1, \ldots, m$.

Fig. 11.13. MaxMinGreedy is a greedy algorithm for solving the max-min version of (MCKP).

The inner loop of Step 3. is executed $O(c)$ times, since the weight sum of the chosen items will be increased by at least one in each iteration. In each step we need to extract the smallest profit which can be done in $O(\log m)$ using a binary heap. Under the natural assumption that the optimal solution value z is much smaller than c, we can maintain a binary array of entries $0, \ldots, c$ in which we mark the current values of $\overline{p}_{ik_i}$. Then the whole loop of Step 3 can be performed in $O(c)$ time, since we only need to move forward in the auxiliary array.

Hifi and Sadfi [224] reached the same time complexity $O(nc)$ but with a smaller space bound through a delayed evaluation of (11.37). Initally the classes $\overline{N}_i$ are only evaluated up to $d = 0$. When a class h is selected in Step 3 of algorithm MaxMinGreedy the corresponding value of d_h is incremented, and (11.37) is solved for the increased value of d_h.

The LP-relaxation C(KSP) of (11.36) can be solved in linear time through a median search algorithm presented by Kuno, Konno and Zemel [290]. The algorithm is based on a decomposition into m subproblems corresponding to the classes N_i, $i = 1, \ldots, m$ as follows

$$\begin{aligned} (\mathrm{KSP}_i) \qquad & \text{maximize } z_i(c_i) = \sum_{j \in N_i} p_{ij} x_{ij} \\ & \text{subject to } \sum_{j \in N_i} w_{ij} x_{ij} \leq c_i, \\ & 0 \leq x_{ij} \leq 1, \quad j \in N_i, \end{aligned} \tag{11.39}$$

where c_i is the capacity associated with class N_i. In this way C(KSP) can be seen as a splitting of the capacity c in m parts so as to maximize the overall objective

$$\text{maximize} \min_{i=1,\ldots,m} z_i(c_i)$$

$$\text{subject to} \sum_{i=1}^{m} c_i \leq c. \tag{11.40}$$

Technically, it is more convenient to work on the inverse problem of (11.39) given by

$$\begin{aligned} (\text{IKSP}_i) \qquad & \text{minimize } c_i(z) = \sum_{j \in N_i} w_{ij} x_{ij} \\ & \text{subject to} \sum_{j \in N_i} p_{ij} x_{ij} \geq z, \\ & 0 \leq x_{ij} \leq 1, \quad j \in N_i. \end{aligned} \tag{11.41}$$

Note that each problem $c_i(z)$ can be solved in $O(|N_i|)$ time, since it is a linear (KP) in minimization form (see also Section 13.3.3). Using the above subproblems we may reformulate $C(KSP)$ as follows

$$\begin{aligned} & \text{maximize } z \\ & \text{subject to} \sum_{i=1}^{m} c_i(z) \leq c, \\ & z \geq 0, \end{aligned} \tag{11.42}$$

where we only have one parameter z. The optimal solution value to $C(\text{KSP})$ will be denoted by z^*.

Since each of the functions $c_i(z)$ is piecewise linear and convex, we may use binary search over the parameter z in (11.42) to find the optimal solution value z^* to $C(\text{KSP})$. For a given value of z we simply solve, in linear time, the m problems $\text{IKSP}_i(z)$. Due to the convexity we have the following implications

$$\sum_{i=1}^{m} c_i(z) = c \quad \Rightarrow \quad z = z*, \tag{11.43}$$

$$\sum_{i=1}^{m} c_i(z) < c \quad \Rightarrow \quad z < z^*, \tag{11.44}$$

$$\sum_{i=1}^{m} c_i(z) > c \quad \Rightarrow \quad z > z^*, \tag{11.45}$$

making it possible to use binary search to determine z^*. The time complexity is further decreased by eliminating some variables in each iteration. Consider a class N_i and assume that $c_i(z)$ was solved for the current value of z. Then we have the following *monotonicity properties*

1. If $x_{ik} = 1$ in $c_i(z)$ and $z < z^*$, then $x_{ij} = 1$ for every $j \in N_i$ with $\frac{p_{ij}}{w_{ij}} \geq \frac{p_{ik}}{w_{ik}}$.

2. If $x_{ik} = 0$ in $c_i(z)$ and $z > z^*$, then $x_{ij} = 0$ for every $j \in N_i$ with $\frac{p_{ij}}{w_{ij}} \leq \frac{p_{ik}}{w_{ik}}$.

To simplify notation we may assume that $p_{ij}/w_{ij} \neq p_{ik}/w_{ik}$ for $j,k \in N_i$ when $j \neq k$. The general case can be analyzed in a similar way. The binary search alogorithm, named KspPartition, is outlined in Figure 11.14. In each iteration, we determine $\tilde{z}$ as a weighted median of the efficiencies in the classes $N_1,\ldots,N_m$. Using this value of $\tilde{z}$ the problems (11.41) are solved and the corresponding capacity sum $\sum_{i=1}^{m} c_i(\tilde{z})$ is calculated. If $\tilde{z}$ is optimal according to (11.43) the algorithm terminates. Otherwise it eliminates some of items in the classes $N_1,\ldots,N_m$ by fixing their values at 0 or 1, and the process is repeated. Items with fixed decision values are removed from the problem. P_i refers to profit sum of the items fixed at 1 in class N_i. In each step of the algorithm, N_i refers to the set of remaining items. If $|N_i| = 1$, the class is eliminated since $c_i(z)$ is a simple linear function.

Algorithm KspPartition:

0. Let $P_i := 0$ be the profit of variables fixed at 1 in class N_i, $i = 1,\ldots,m$.
1. For every remaining class N_i, $i = 1,\ldots,m$,
   compute r_i as the median of the set $\{\frac{p_{ij}}{w_{ij}} \mid j \in N_i\}$.
   $L := \{j \in N_i \mid \frac{p_{ij}}{w_{ij}} < r_i\}$
   $G := \{j \in N_i \mid \frac{p_{ij}}{w_{ij}} \geq r_i\}$
   $z_i := \sum_{j \in G} p_{ij} + P_i$.
   $Z_i := \{z_i, z_i, \ldots, z_i\}$ where $|Z_i| = |N_i|$.
2. Compute $\tilde{z}$ as the median of the set $\{\cup_{i=1}^{m} Z_i\}$
3. Solve the problems $c_i(\tilde{z})$ for $i = 1,\ldots,m$.
   If $\tilde{z}$ is optimal according to test (11.43) then stop.
4. For every class $i = 1,\ldots,m$ eliminate variables as follows:
   If (11.44) is satisfied and $\tilde{z} \geq z_i$ then set $x_{ik} := 1$ for $k \in G$.
   If (11.45) is satisfied and $\tilde{z} \leq z_i$ then set $x_{ik} := 0$ for $k \in L$.
   Remove fixed items from N_i and update P_i accordingly.
5. If $m = 0$, i.e. no classes are remaining, then
   find z^* by solving the current linear equation.
   Otherwise go to Step 1.

Fig. 11.14. The median search algorithm for solving the (KSP) is a generalization of the Split algorithm from Section 3.1.

Kuno, Konno and Zemel [290] proved the following time complexity

Proposition 11.10.1 *Algorithm* KspPartition *solves the* $C(KSP)$ *in* $O(n)$ *time*

Proof. Each iteration of Steps 1 to 5 can be performed in linear time. Moreover, we will show that a constant fraction of the variables is eliminated in each iteration of Step 4.

If $\sum_{i=1}^{m} c_i(\tilde{z}) < c$ and $\tilde{z} \geq z_i$, then $z^* \geq z_i$. The variable $k \in N_i$ having efficiency $p_{ik}/w_{ik} = r_i$ must have $x_{ik} = 1$ in $\mathrm{IKSP}_i(z^*)$. Due to the monotonicity properties outlined above, variables in G can be fixed to 1. In a similiar way if $\sum_{i=1}^{m} c_i(\tilde{z}) > c$ and $\tilde{z} \leq z_i$, then $z^* \leq z_i$ and the variables in L can be fixed to 0. In either case at least $n_i/2$ variables can be eliminated. The number of classes for which $\tilde{z} \geq z_i$ is at least half of the total number of classes, and similarly the number of classes for which $\tilde{z} \leq z_i$ is at least half of the total. Thus in each iteration we will remove at least $1/4$ of the variables. Hence, the running time will be of magnitude $n + \left(\frac{3}{4}\right) n + \left(\frac{3}{4}\right)^2 n + \ldots \leq 4n$, which gives the stated. □

Computational results for exact algorithms are presented in Hifi and Sadfi [224]. Heuristic approaches are considered in Yamada and Futakawa [490] and in Hifi, Sadfi and Sbihi [225]. The latter heuristic is based on so-called single-depth and multi-depth tabu search methods. Reduction techniques for the (KSP) are presented in Yamada and Futakawa [490].

12. The Quadratic Knapsack Problem

In all the variants of the knapsack problems considered so far the profit of choosing a given item was independent of the other items chosen. In many real life applications as well as in problems with roots in graph theory it is natural to assume that the profit of a packing also should reflect how well the given items fit together. One possible formulation of such an interdependence is the *quadratic knapsack problem* (QKP) in which an item has a corresponding profit and an additional profit is redeemed if the item is selected together with another item. (QKP) was first introduced by Gallo, Hammer and Simeone [160] and has been studied intensively in the last decade due to its simple structure and challenging difficulty.

12.1 Introduction

To define (QKP) more formally assume that n items are given, item j having a positive integer weight w_j, and a limit on the total weight chosen is given by a positive integer knapsack capacity c. In addition we are given an $n \times n$ nonnegative integer *profit matrix* $P = (p_{ij})$, where p_{jj} is the profit achieved if item j is selected and $p_{ij} + p_{ji}$ is a profit achieved if both items i and j are selected. (QKP) calls for selecting an item subset whose overall weight does not exceed the knapsack capacity, so as to maximize the overall profit. Recall that the set of items is $N := \{1,\dots,n\}$ and for notational convenience we will in this chapter write sums over the set N instead of explicitly summing over $1,\dots,n$.

By introducing a binary variable x_j equal to 1 if item j is selected and 0 otherwise, the problem has the following integer programming formulation:

$$\begin{aligned} \text{(QKP)} \qquad & \text{maximize} \sum_{i\in N}\sum_{j\in N} p_{ij}x_ix_j \\ & \text{subject to} \sum_{j\in N} w_jx_j \le c, \\ & \qquad x_j \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.1}$$

As before, we assume without loss of generality that

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

$$\max_{j \in N} w_j \leq c < \sum_{j \in N} w_j, \tag{12.2}$$

and that the profit matrix is symmetric, i.e., $p_{ij} = p_{ji}$ for all $i, j \in N$. Notice, that if negative weights $w_j < 0$ are present, we may flip variable x_j to $1 - x_j$. If $w_j > c$ we may fix $x_j = 0$, and if $w_j = c$ then we may decompose the problem. Hence, normally $0 \leq w_j < c$. If $p_{ij} \geq 0$ for all coefficients $i \neq j$ (QKP) is denoted the *supermodular knapsack problem*. In the sequel we will rely on the stronger assumption that all coefficients $p_{ij} \geq 0$, which is not made without loss of generality. Where upper bounds and other results are valid for a more general model, it will be stated in the text.

One may give several graph-theoretic interpretation to (QKP): Given a complete undirected graph on node set N, where each node j has a profit p_{jj} and weight w_j and each edge (i, j) has a profit $p_{ij} + p_{ji}$, select a node subset $S \subseteq N$ whose overall weight does not exceed c so as to maximize the overall profit, given by the sum of the profits of the nodes in S and of the edges with both endpoints in S. It is then easy to see that (QKP) is also a generalization of the CLIQUE problem. This latter problem, in its recognition version, calls for checking whether, for a given positive integer k, a given undirected graph $G = (V, E)$ contains a complete subgraph on k nodes. A possible optimization version of CLIQUE is given by the so-called *dense subgraph problem*, in which one wants to select a node subset $S \subseteq V$ of cardinality $|S| = k$ such that the subgraph of G induced by S contains as many edges as possible. This problem can be modeled as (QKP) by setting $n := |V|$, $c := k$, and $w_j := 1$ for $j \in N$. Moreover we set $p_{ij} := p_{ji} := 1$ if $(i, j) \in E$ and $p_{ij} := p_{ji} := 0$ otherwise, for $i, j \in N$. Note that in this case the knapsack constraint reduces to a cardinality constraint, which will be satisfied with equality by the optimal solution. Clearly, the answer to CLIQUE is positive if and only if the optimal solution of this (QKP) has value $k(k-1)$. The most famous optimization version of CLIQUE, called *max clique*, calls for an induced complete subgraph with a maximum number of nodes. This latter problem can be solved through a (QKP) algorithm by using binary search with c between 1 and n.

As opposed to most of the knapsack problems considered so far, (QKP) is $\mathcal{NP}$-hard in the strong sense, which can be seen by reduction from the clique problem. Hence, no dynamic programming algorithm with pseudopolynomial running time can exist unless $\mathcal{P} = \mathcal{NP}$.

A special version of (QKP) appears when restricting the problem to a *diagonal profit matrix* P, such that $p_{ij} = 0$ for $i \neq j$. If the variables x_j are binary, this problem becomes the ordinary (KP). The *integer diagonal quadratic knapsack problem*, where variables may take on any integer value between a lower and an upper bound, is considered by Bretthauer, Shetty and Syam [50].

As one might expect, due to its generality, (QKP) has a wide spectrum of applications. Witzgall [484] presented a problem which arises in telecommunications when a number of sites for satellite stations have to be selected, such that the global traffic

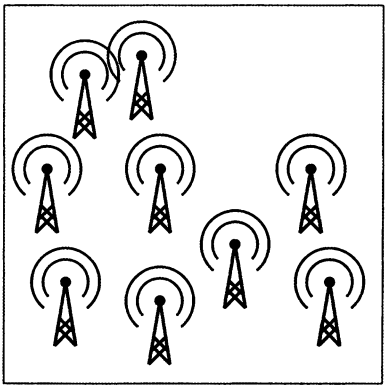


Fig. 12.1. A quadratic knapsack problem appears in telecommunications when a number of sites for satellite stations have to be selected, such that the global traffic between these stations is maximized and a budget constraint is respected.

between these stations is maximized and a budget constraint is respected. This problem appears to be a (QKP) as illustrated in Figure 12.1. Similar models arise when considering the location of airports, railway stations or freight handling terminals [406]. Johnson, Mehrotra and Nemhauser [253] mention a compiler design problem which may be formulated as a (QKP), as described in [219]. Dijkhuizen and Faigle [106] and Park, Lee and Park [370] consider the weighted maximum b-clique problem. If all edge weights are nonnegative this problem is the special case of (QKP) arising when $w_j = 1$ for $j \in N$ and $b = c$. Finally, (QKP) appears as the column generation subproblem when solving the graph partitioning problem described in Johnson, Mehrotra and Nemhauser [253].

In the following Section 12.2 we will give a survey of the most important upper bounds for (QKP). Techniques for variable reduction are presented in Section 12.3, and branch-and-bound algorithms are discussed in Section 12.4. Next, in Section 12.5, we will go into details with a branch-and-bound algorithm presented by Caprara, Pisinger and Toth [69] which has been able to solve some of the largest (QKP) problems in the literature. The following two sections 12.6 and 12.7 deal with heuristics and approximation algorithms. Finally, in Section 12.8, a number of computational experiments are presented showing the performance of the Quad-Knap algorithm, and also showing the quality of all the presented upper bounds. This chapter is partially based on the survey paper [391].

12.2 Upper Bounds

Numerous upper bounds have been presented for the quadratic knapsack problem during the last two decades. The bounds are based on a variety of techniques including: linearisation, Lagrangian relaxation, derivation of upper planes, semidefinite

relaxations and reformulation techniques. In the following we will give a quite detailed survey of the most important bounds.

Since the upper bounds typically are used in a branch-and-bound algorithm, one must weight tightness against computational time. Using very tight bounds will result in few nodes visited, but the time consumption may still become large due to the complexity of computing the bounds. On the other hand, less tight but quickly computable bounds will result in a large search tree, but the time consumption may be reasonable if very little time is spent at each node. The ideal approach to use, depends on the nature of the problem considered and the instances being solved.

12.2.1 Continuous Relaxation

By relaxing the bounds on x_j to $0 \leq x_j \leq 1$ we get a *continuous quadratic knapsack problem*. It would be misleading to name it *linear quadratic knapsack problem* as the objective function is not linear. Due to the quadratic objective the problem cannot be solved just through linear programming.

The difficulty of solving the continuous quadratic knapsack problem depends largely on the nature of the profit matrix P. If the matrix is positive semidefinite $P \succeq 0$ the continuous quadratic knapsack problem can be solved efficiently. We will go into details with semidefinite programming in Section 12.2.7. If P has negative eigenvalues, a non-convex quadratic programming problem appears. In this situation the objective function may have more than one local maximum (instances can be constructed with an exponential number of local maxima) and possibly forcing a solution algorithm to visit all the local maxima.

By Lagrangian relaxing the capacity constraint, and solving the Lagrangian dual problem, an upper bound can be reached in polynomial time. As will be shown in Section 12.2.2, the bound obtained this way is equivalent to the bound obtained by solving the continuous quadratic knapsack problem.

12.2.2 Bounds from Lagrangian Relaxation of the Capacity Constraint

Chaillou, Hansen and Mahieu [75] Lagrangian relaxed the capacity constraint in (QKP) using multiplier $\lambda \geq 0$, getting the problem

$$\begin{aligned} &\text{maximize} \sum_{i \in N} \sum_{j \in N} p_{ij} x_i x_j - \lambda \left(\sum_{j \in N} w_j x_j - c \right) \\ &\text{subject to } x_j \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.3}$$

By setting

$$\tilde{p}_{ij} := \begin{cases} p_{ij} & \text{if } i \neq j, \\ p_{ij} - \lambda w_j & \text{if } i = j, \end{cases} \tag{12.4}$$

the relaxed problem can be reformulated as a *quadratic 0-1 programming problem* $L_1(\text{QKP},\lambda)$ of the form

$$\begin{aligned} &\text{maximize} \sum_{i\in N}\sum_{j\in N} \tilde{p}_{ij}x_ix_j + \lambda c \\ &\text{subject to } x_j \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.5}$$

Picard and Ratliff [377] showed that the latter problem can be solved in polynomial time, since the matrix $\{\tilde{p}_{ij}\}$ has nonnegative off-diagonal elements. Chaillou, Hansen and Mahieu [75] presented a simple algorithm based on the solution of a *maximum flow problem* in a network (see e.g. Cormen et al [92]). The capacitated network $\mathcal{N} = (V,E,\bar{c})$ is defined with a vertex set $V = \{s,1,\ldots,n,t\}$ where s is the source and t is the sink. The edge set E is defined on $V \times V$ having the capacities

$$\begin{aligned} \bar{c}_{si}(\lambda) &:= \max\left(0, \textstyle\sum_{j\in N} p_{ij} - \lambda w_i\right), \\ \bar{c}_{ij}(\lambda) &:= p_{ij}, \\ \bar{c}_{it}(\lambda) &:= \max\left(0, \lambda w_i - \textstyle\sum_{j\in N} p_{ij}\right). \end{aligned} \tag{12.6}$$

Algorithm ChaillouHansenMahieu

1. *Initialization*
   $\alpha_1 := c - \sum_{j\in N} w_j$, $\alpha_2 := c$, $\beta_1 := \sum_{i\in N}\sum_{j\in N} p_{ij}$, $\beta_2 := 0$
2. *Maximum flow problem*
   Compute $\lambda := (\beta_1 - \beta_2)/(\alpha_2 - \alpha_1)$.
   Construct the network $\mathcal{N} = (V,E,\bar{c})$.
   Determine the maximum flow between s and t.
   Determine the solution value $z(L_1(\text{QKP},\lambda))$ by (12.7).
   Let $\alpha_3\lambda + \beta_3$ be the line in the plane $(\lambda, f(\lambda))$
   with $f(\lambda) = z(L_1(\text{QKP},\lambda))$.
3. *Optimality test*
   if $\alpha_3\lambda + \beta_3 = \alpha_1\lambda + \beta_1$ then λ is optimal, stop.
   if $\alpha_3 < 0$ then
      set $\alpha_1 := \alpha_3$, $\beta_1 := \beta_3$ and go to Step 2.
   else
      set $\alpha_2 := \alpha_3$, $\beta_2 := \beta_3$ and go to Step 2.

Fig. 12.2. The binary search algorithm by Chaillou, Hansen and Mahieu for determining the Lagrangian multiplier λ^* which minimizes the Lagrangian dual.

Assume that the value of the maximum flow from s to t in $\mathcal{N}$ is $\Psi(\mathcal{N})$. Then Chaillou, Hansen and Mahieu [75] showed the following

Proposition 12.2.1 *An optimal solution value to (12.3) is given by*

$$z(L_1(QKP,\lambda)) = \lambda c + \sum_{i\in N} \bar{c}_{si}(\lambda) - \Psi(\mathcal{N}). \tag{12.7}$$

Proof. A cut $c(S,T)$ separates the set of nodes V into two subsets S and T where $s \in S$ and $t \in T$. Let $x_i = 1$ if node i is in set S and $x_i = 0$ if it is in set T. The capacity of a cut is hence $\sum_{i\in N} \bar{c}_{si}(1-x_i) + \sum_{i\in N}\sum_{j\in N} \bar{c}_{ij}x_i(1-x_j) + \sum_{i\in N} \bar{c}_{it}x_i$. Since the maximum flow $\Psi(G)$ equals a minimum cut $c(S,T)$ we obtain

$$\begin{aligned}\Psi(G) &= \min_{x\in\{0,1\}^n} \left\{ \sum_{i\in N} \bar{c}_{si}(1-x_i) + \sum_{i\in N}\sum_{j\in N} \bar{c}_{ij}x_i(1-x_j) + \sum_{i\in N} \bar{c}_{it}x_i \right\} \qquad (12.8)\\ &= \sum_{i\in N} \bar{c}_{si} + \min_{x\in\{0,1\}^n} \left\{ \sum_{i\in N} (\bar{c}_{it} - \bar{c}_{si} + \sum_{j\in N} \bar{c}_{ij})x_i - \sum_{i\in N}\sum_{j\in N} \bar{c}_{ij}x_ix_j \right\}.\end{aligned}$$

We have that

$$\begin{aligned}\bar{c}_{it} - \bar{c}_{si} &= \max\{0, \lambda w_i - \sum_{j\in N} p_{ij}\} - \max\{0, \sum_{j\in N} p_{ij} - \lambda w_i\} \qquad (12.9)\\ &= \max\{0, \lambda w_i - \sum_{j\in N} p_{ij}\} + \min\{0, \lambda w_i - \sum_{j\in N} p_{ij}\}\\ &= \lambda w_i - \sum_{j\in N} p_{ij} = \lambda w_i - \sum_{j\in N} \bar{c}_{ij},\end{aligned}$$

which implies that $\bar{c}_{it} - \bar{c}_{si} + \sum_{j\in N} \bar{c}_{ij} = \lambda w_i$. This gives

$$\begin{aligned}\Psi(G) &= \sum_{i\in N} \bar{c}_{si} + \min_{x\in\{0,1\}^n} \left\{ \sum_{i\in N} \lambda w_i x_i - \sum_{i\in N}\sum_{j\in N} \bar{c}_{ij}x_ix_j \right\} \qquad (12.10)\\ &= \sum_{i\in N} \bar{c}_{si} - z(L_1(\mathrm{QKP},\lambda)) + \lambda c.\end{aligned}$$

Rearranging the terms gives the stated. □

The Lagrangian dual problem of (12.3) is

$$\min_{\lambda\geq 0} L_1(\mathrm{QKP},\lambda). \tag{12.11}$$

Let U_{CHM} denote the bound corresponding to the optimal solution of (12.11). Chaillou, Hansen and Mahieu proved that the Lagrangian dual is a piecewise linear function of λ with at most n linear segments. This observation can be used to construct a simple binary search algorithm for determining the optimal choice λ^* of Lagrangian multiplier. The algorithm by Chaillou, Hansen and Mahieu is depicted in Figure 12.2. Since there are at most n linear segments, we will use no more than $O(n)$ iterations in the main loop, each time solving a maximum flow problem on a graph with

$n+2$ vertices and $2n+n^2$ edges. Using Karzanovs algorithm [260] takes $O(|V|^3)$ time for each maximum flow problem, hence in total $O(|V|^4)$ time is used. Gallo, Grigoriadis and Tarjan [159] further improved the complexity of solving the Lagrangian dual problem by taking advantage of the similarity of successive maximum flow problems.

Using the results by Geoffrion [172], we note that the bound obtained from the Lagrangian dual problem is equivalent to the bound obtained through continuous relaxation described in Section 12.2.1. But the first-mentioned can be computed efficiently as described above.

12.2.3 Bounds from Upper Planes

Gallo, Hammer and Simeone [160] presented the first bounds for (QKP) using the concept of *upper plane*. An upper plane (or *linear majorization function*) is a linear function g satisfying $g(x) \geq \sum_{i\in N}\sum_{j\in N} p_{ij}x_ix_j$ for any feasible solution $x=(x_1,\ldots,x_n)$ of (12.1). The actual upper planes proposed in [160] are of the form $\sum_{j\in N}\pi_j x_j$, for $j \in N$ leading to the following relaxed optimization problem

$$\begin{aligned} &\text{maximize} \sum_{j\in N}\pi_j x_j \\ &\text{subject to} \sum_{j\in N} w_j x_j \leq c, \\ &\qquad x_j \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.12}$$

The problem is recognized as an ordinary (KP) since we have a linear objective function $g(x)$ subject to a linear capacity constraint. Gallo, Hammer and Simeone consider four different expressions of π_j, leading to the bounds $U^1_{\text{GHS}}, U^2_{\text{GHS}}, U^3_{\text{GHS}}, U^4_{\text{GHS}}$. The bound U^1_{GHS} is obtained by using the upper plane

$$\pi^1_j := \sum_{i\in N} p_{ij}, \tag{12.13}$$

which obviously is valid since $\sum_{j\in N}\sum_{i\in N} p_{ij}\bar{x}_i\bar{x}_j \leq \sum_{j\in N}\left(\sum_{i\in N} p_{ij}\right)\bar{x}_j$ for any feasible solution vector $\bar{x}$.

Bound U^2_{GHS} is based on the upper plane

$$\pi^2_j := \max\left\{\sum_{i\in N} p_{ij}\bar{x}_i \,\middle|\, \sum_{i\in N}\bar{x}_i \leq k, \bar{x}_i \in \{0,1\} \text{ for } i \in N\right\}, \tag{12.14}$$

where k is the maximum cardinality of a feasible (QKP) solution as defined in (5.14) (see Section 5.1.1). In other words π^2_j is the sum of the k biggest profits among $\{p_{1j},\ldots,p_{nj}\}$. We notice that the upper plane is valid since $\sum_{j\in N}\left(\sum_{i\in N} p_{ij}\bar{x}_i\right)\bar{x}_j \leq \sum_{j\in N}\pi^2_j\bar{x}_j$ for any feasible solution $\bar{x}$.

The next bound, U^3_{GHS}, is obtained by using the upper plane

$$\pi^3_j := \max\left\{\sum_{i\in N} p_{ij}\bar{x}_i \,\middle|\, \sum_{i\in N} w_i\bar{x}_i \le c, 0 \le \bar{x}_i \le 1 \text{ for } i \in N\right\}, \tag{12.15}$$

which appears by noting that $\sum_{i\in N} p_{ij}x_i$ subject to the constraint $\sum_{i\in N} w_i x_i \le c, x_i \in \{0,1\}$ does not exceed π^3_j.

Finally, U^4_{GHS} is derived by setting

$$\pi^4_j := \max\left\{\sum_{i\in N} p_{ij}\bar{x}_i \,\middle|\, \sum_{i\in N} w_i\bar{x}_i \le c, \bar{x}_i \in \{0,1\} \text{ for } i \in N\right\}. \tag{12.16}$$

The validity of this bound is checked as above.

Due to the solution of the knapsack problem (12.12) none of the bounds U^1_{GHS}, U^2_{GHS}, U^3_{GHS}, U^4_{GHS} have polynomial time bounds. However, if we solve the continuous relaxation of (12.12) we obtain the weaker bounds U^{1c}_{GHS}, U^{2c}_{GHS}, U^{3c}_{GHS}, U^{4c}_{GHS}. Of these bounds U^{1c}_{GHS}, U^{2c}_{GHS}, U^{3c}_{GHS} can be obtained in $O(n^2)$ time, while it is $\mathcal{NP}$-hard to derive U^{4c}_{GHS} due to the solution of (12.16). Caprara, Pisinger and Toth [69] noticed that coefficients π^2_j, π^3_j and π^4_j can be improved by forcing $\bar{x}_j = 1$ in the computation of the different values of π_j. We will denote these bounds by $\overline{U}^2_{\text{GHS}}$, $\overline{U}^3_{\text{GHS}}$ and $\overline{U}^4_{\text{GHS}}$.

It can easily be verified that $U^4_{\text{GHS}} \le U^3_{\text{GHS}}$ and $U^4_{\text{GHS}} \le U^2_{\text{GHS}}$. Moreover $U^3_{\text{GHS}} \le U^1_{\text{GHS}}$ and $U^2_{\text{GHS}} \le U^1_{\text{GHS}}$. Hence U^4_{GHS} is the tightest of the bounds. No dominance exists between U^2_{GHS} and U^3_{GHS} as illustrated in Figure 12.3.

Instance 1:
$n = 3, c = 3, w = (1,2,2), p = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 2 \end{pmatrix}$

Instance 2:
$n = 2, c = 3, w = (2,2), p = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$

Fig. 12.3. An example showing that no dominance exists between U^2_{GHS} and U^3_{GHS}. In instance 1 we find $U^2_{\text{GHS}} = 4$ and $U^3_{\text{GHS}} = 3$ hence $U^2_{\text{GHS}} \not\le U^3_{\text{GHS}}$. The opposite situation appears in instance 2, where we find $U^2_{\text{GHS}} = 2$ and $U^3_{\text{GHS}} = 3$ hence $U^3_{\text{GHS}} \not\le U^2_{\text{GHS}}$.

Based on several computational experiments, Gallo, Hammer and Simeone argued that the upper bound U^{3c}_{GHS} gives the best trade-off between tightness and computational effort in a branch-and-bound algorithm.

12.2.4 Bounds from Linearisation

An obvious approach for deriving an upper bound for (QKP) is to linearize the quadratic term, and then solve the LP-relaxation of the problem. Billionnet and Calmels [37] presented a bound based on this principle, by introducing variables

y_{ij} which attain the value 1 if and only if $x_i = 1$ and $x_j = 1$. We may formulate this equivalence by the constraints

$$y_{ij} \leq x_i, \qquad y_{ij} \leq x_j, \qquad x_i + x_j \leq 1 + y_{ij}, \tag{12.17}$$

which leads to the following integer linear programming model

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in N} \sum_{j \in N} p_{ij} y_{ij} \\ \text{subject to} \quad & \sum_{j \in N} w_j x_j \leq c, \\ & y_{ij} \leq x_i, && i,j \in N, \\ & y_{ij} \leq x_j, && i,j \in N, \\ & x_i + x_j \leq 1 + y_{ij}, && i,j \in N, \\ & x_j, y_{ij} \in \{0,1\}, && i,j \in N. \end{aligned} \tag{12.18}$$

By using the fact that $y_{ij} = y_{ji}$ we may rewrite the model to

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in N} \sum_{j \in N, j > i} 2 p_{ij} y_{ij} \\ \text{subject to} \quad & \sum_{j \in N} w_j x_j \leq c, \\ & y_{ij} \leq x_i, && i,j \in N, i < j, \\ & y_{ij} \leq x_j, && i,j \in N, i < j, \\ & x_i + x_j \leq 1 + y_{ij}, && i,j \in N, i < j, \\ & x_j, y_{ij} \in \{0,1\}, && i,j \in N, i < j. \end{aligned} \tag{12.19}$$

The LP-relaxation of the latter formulation leads to tighter upper bounds than (12.18) since it implicitly contains the constraints $y_{ij} = y_{ji}$.

Since $p_{ij} \geq 0$ the constraints $x_i + x_j - 1 \leq y_{ij}$ are not strictly necessary in (12.19). Relaxing the integrality constraints to $0 \leq x_i \leq 1$ and $y_{ij} \geq 0$ we get the model

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in N} \sum_{i \in N \setminus \{j\}} p_{ij} y_{ij} + \sum_{j \in N} p_{jj} x_j \\ \text{subject to} \quad & \sum_{j \in N} w_j x_j \leq c, \\ & y_{ij} \leq x_i, && j \in N, \\ & y_{ij} \leq x_j, && i \in N, \\ & x_i + x_j \leq 1 + y_{ij}, && i,j \in N, \\ & 0 \leq x_j \leq 1, && j \in N, \\ & y_{ij} \geq 0, && i,j \in N. \end{aligned} \tag{12.20}$$

Solving (12.20) leads to the bound U^1_{BC}, which according to Billionnet and Calmels is of quite bad quality.

In order to tighten the above formulation a number of additional constraints are added to the model. By multiplying the capacity constraint

$$\sum_{i \in N} w_i x_i \leq c, \tag{12.21}$$

with x_j for each $j \in N$ and replacing x_j^2 by x_j (note that $x_j \in \{0,1\}$) we obtain n new constraints

$$\sum_{i \in N \setminus \{j\}} w_i x_i x_j \leq (c - w_j) x_j, \quad j \in N. \tag{12.22}$$

Although these constraints are redundant for the integer programming formulation, they may tighten the LP-relaxation (12.20). Such constraints are also used in Helmberg, Rendl and Weismantel [219], and Caprara, Pisinger and Toth [69]. They may be seen as application of a general procedure proposed by Adams and Sherali [3] and further studied by Lovász and Schrijver [308] and Balas, Ceria and Cornuéjols [19].

Further, for each three indices $i \neq j \neq k$ we derive a *Chvatal-Gomory cut* by adding together three of the constraints $x_i + x_j \leq 1 + y_{ij}$ in (12.18) as

$$\begin{aligned} x_i + x_j - y_{ij} &\leq 1, \\ x_j + x_k - y_{jk} &\leq 1, \\ x_k + x_i - y_{ki} &\leq 1, \end{aligned} \tag{12.23}$$

getting the inequality $2x_i + 2x_j + 2x_k - y_{ij} - y_{jk} - y_{ki} \leq 3$. Dividing by two, and rounding down all coefficients one gets the inequality $x_i + x_j + x_k - y_{ij} - y_{jk} - y_{ki} \leq 1$.

This leads to the following formulation

$$\text{maximize} \sum_{j \in N} \sum_{i \in N \setminus \{j\}} p_{ij} y_{ij} + \sum_{j \in N} p_{jj} x_j$$

$$\text{subject to} \sum_{j \in N} w_j x_j \leq c, \tag{12.24}$$

$$y_{ij} \leq x_i, \qquad i,j \in N, \tag{12.25}$$

$$y_{ij} \leq x_j, \qquad i,j \in N, \tag{12.26}$$

$$x_i + x_j \leq 1 + y_{ij}, \qquad i,j \in N, \tag{12.27}$$

$$0 \leq x_j \leq 1, \qquad j \in N, \tag{12.28}$$

$$y_{ij} \geq 0, \qquad i,j \in N, \tag{12.29}$$

$$\sum_{i \in N \setminus \{j\}} w_i y_{ij} \leq (c - w_j) x_j, \qquad j \in N, \tag{12.30}$$

$$x_i + x_j + x_k - y_{ij} - y_{jk} - y_{ki} \leq 1, \quad i < j < k. \tag{12.31}$$

Solving the model gives the bound U_{BC}^2 which according to Billionnet and Calmels is considerably tighter than U_{BC}^1. The model has $O(n^2)$ variables and $O(n^3)$ constraints, so for $n = 20$ more than 1500 constraints are present. This means that it is quite time consuming to derive the bound for large-sized instances. Billionnet and Calmels

hence propose to solve the model defined on (12.24), (12.28), (12.29) and (12.30) only. This model has $O(n)$ constraints, hence being easy to solve. Then, a search is performed which finds all constraints among (12.25), (12.26), (12.27) and (12.31) which are not satisfied. These constraints are added to the model, and the process is repeated progressively. According to Billionnet and Calmels this means that a problem with $n = 20$ may be solved with typically 300 constraints instead of the previously mentioned 1500.

12.2.5 Bounds from Reformulation

The bound by Caprara, Pisinger and Toth [69] can be described within the framework of upper planes as follows. First we derive a new upper plane as

$$\pi_j^5 := p_{jj} + \max\left\{ \sum_{i \in N\setminus\{j\}} p_{ij}\bar{x}_i \;\middle|\; \begin{array}{l} \sum_{i\in N\setminus\{j\}} w_i\bar{x}_i \le (c - w_j), \\ \bar{x}_i \in \{0,1\} \text{ for } i \in N\setminus\{j\} \end{array} \right\}. \tag{12.32}$$

Then an upper bound U_{CPT}^1 is derived as the optimal solution value to (12.12). If we relax the integrality constraints in the above subproblems (12.32) we obtain the bound U_{CPT}^{1c} which can be derived in $O(n^2)$ time by solving the *n linear knapsack problems* for $j = 1,\ldots,n$

$$\begin{aligned} \text{maximize } \pi_j^{5*} &= p_{jj} + \sum_{i\in N\setminus\{j\}} p_{ij}\bar{x}_i \\ \text{subject to } & \sum_{i\in N\setminus\{j\}} w_i\bar{x}_i \le (c - w_j), \\ & 0 \le \bar{x}_i \le 1, \quad i \in N\setminus\{j\}, \end{aligned} \tag{12.33}$$

and solving the linear knapsack problem (LKP) associated with

$$\begin{aligned} \text{maximize } & \sum_{j\in N} \pi_j^{5*}\bar{x}_j \\ \text{subject to } & \sum_{j\in N} w_j\bar{x}_j \le c, \\ & 0 \le \bar{x}_j \le 1, \quad j \in N. \end{aligned} \tag{12.34}$$

Caprara, Pisinger and Toth [69] further strengthened the bound by noting that the objective function can be reformulated as

$$\sum_{i\in N}\sum_{j\in N} p_{ij}x_ix_j = \sum_{i\in N}\sum_{j\in N}(p_{ij} + \lambda_{ij})x_ix_j, \tag{12.35}$$

for any matrix $\Lambda = \{\lambda_{ij}\}$ satisfying that $\lambda_{ij} = -\lambda_{ji}$, i.e. Λ should be a *skew-symmetric matrix*. Let QKP(Λ) denote the problem

$$\begin{aligned} \text{maximize} \quad & \sum_{i\in N}\sum_{j\in N}(p_{ij}+\lambda_{ij})x_i x_j \\ \text{subject to} \quad & \sum_{j\in N} w_j x_j \le c, \\ & x_j \in \{0,1\}, \quad j \in N, \end{aligned} \tag{12.36}$$

and $U^{1c}_{\text{CPT}}(\Lambda)$ the corresponding upper bound obtained by solving (12.33) and (12.34). In order to obtain the tightest bound we solve the Lagrangian dual problem

$$U^2_{\text{CPT}} := \min_{\Lambda \text{ skew-symmetric}} U^{1c}_{\text{CPT}}(\Lambda). \tag{12.37}$$

The latter problem may be solved through subgradient optimization (see Sections 3.8 and 9.2) leading to the bound $\hat{U}^2_{\text{CPT}}$ for some near-optimal matrix Λ of Lagrangian multipliers. Obviously $U^2_{\text{CPT}} \le \hat{U}^2_{\text{CPT}} \le U^{1c}_{\text{CPT}}(0)$.

In the subgradient optimization, Caprara, Pisinger and Toth initially set the step size parameter μ to 1, and halved it if the upper bound did not decrease within 20 consecutive iterations. The tolerance ε was set to 10^{-6}. The number of iterations was in each case limited by $200+n$, since it was observed that afterwards no substantial improvement occurs. The overall complexity of computing $\hat{U}^2_{\text{CPT}}$ is therefore $O(n^3)$.

Corresponding ILP-Formulation. In order to determine the relative tightness of bound U^1_{CPT} and U^2_{CPT} compared to previously presented bounds, Caprara, Pisinger and Toth [69] consider the ILP-models corresponding to the bounds.

Bound U^1_{CPT} may be seen as an extension of model (12.18) to which the redundant capacity constraints (12.22) have been added, and the constraints (12.17) have been replaced by

$$y_{ij} \le x_j, \qquad y_{ij} = y_{ji}. \tag{12.38}$$

Notice that the last constraints in (12.17) are not strictly necessary. These constraints could be used to tighten the LP-relaxation of the model, but they cannot be handled by the upper plane algorithm used for solving the LP-relaxation. In this way we reach the ILP reformulation of (QKP):

$$\text{maximize} \quad \sum_{j\in N}\sum_{i\in N\setminus\{j\}} p_{ij}y_{ij} + \sum_{j\in N} p_{jj}x_j \tag{12.39}$$

$$\text{subject to} \quad \sum_{j\in N} w_j x_j \le c, \tag{12.40}$$

$$\sum_{i\in N\setminus\{j\}} w_i y_{ij} \le (c-w_j)x_j, \qquad j \in N, \tag{12.41}$$

$$0 \le y_{ij} \le x_j \le 1, \qquad i,j \in N, j \ne i, \tag{12.42}$$

$$y_{ij} = y_{ji}, \qquad i, j \in N, j > i, \tag{12.43}$$

$$x_j, y_{ij} \in \{0,1\}, \qquad i, j \in N, j \neq i. \tag{12.44}$$

The reason for an explicit use of two distinct variables y_{ij} and y_{ji}, linked by equality constraints (12.43) is motivated by the fact that if equations (12.43) are removed and (12.44) LP-relaxed, the resulting problem (12.39)–(12.42) can be solved in a very effective way.

For each $j \in N$, the variables $y_{ij}, i \in N \setminus \{j\}$, besides having a lower bound of 0 and a variable upper bound of x_j, appear only in constraint (12.41) associated with j, and in the objective function. Hence, if variable x_j is fixed to value $\bar{x}_j$ for all $j \in N$, the relaxed problem decomposes into n independent subproblems, one for each $j \in N$, of the form (12.32).

The ILP formulation can also be used to characterize the bounds U^1_{GHS} to U^4_{GHS} by Gallo, Hammer and Simeone. It is immediate to see that coefficients π^2_j, π^3_j and π^4_j can be improved by forcing $\bar{x}_j = 1$ in the computation of π_j. In this case, the upper bounds coincide, respectively, with the following formulations:

U^1_{GHS} corresponds to the solution of (12.39), (12.40), (12.42), and (12.44).

U^{1c}_{GHS} corresponds to the solution of (12.39), (12.40) and (12.42).

U^2_{GHS} corresponds to the solution of (12.39), (12.40), (12.42), and (12.44) with the additional constraints $\sum_{i \in N \setminus \{j\}} y_{ij} \leq k$ for $j \in N$.

U^{2c}_{GHS} corresponds to the solution of (12.39), (12.40), (12.42), with the additional constraints $\sum_{i \in N \setminus \{j\}} y_{ij} \leq k$ for $j \in N$.

U^{3c}_{GHS} corresponds to the solution of (12.39)–(12.42), if the continuous relaxation of the final (KP) is solved.

U^4_{GHS} corresponds to the solution of (12.39)–(12.42) and (12.44).

Instead of removing constraints (12.43), we may obtain a tighter bound through Lagrangian relaxation. We introduce a matrix $\Lambda = (\lambda_{ij})$, where, for $i, j \in N, j > i, \lambda_{ij}$ is the *Lagrangian multiplier* associated with the corresponding equation in (12.43) and, for notational convenience, $\lambda_{ji} := -\lambda_{ij}$. Accordingly, the Lagrangian modified objective function reads:

$$\text{maximize } z(\text{QKP}(\Lambda)) = \sum_{i \in N} \sum_{j \in N} \hat{p}_{ij} y_{ij} \tag{12.45}$$

where

$$\hat{p}_{ij} := \begin{cases} p_{ij} + \lambda_{ij} & \text{if } i \neq j, \\ p_{ij} & \text{if } i = j, \end{cases} \tag{12.46}$$

is the *Lagrangian profit* associated with variable y_{ij}. The corresponding Lagrangian relaxed problem is given by (12.45) subject to (12.40)–(12.42) and (12.44). For

a given Λ, the continuous relaxation of this problem can be solved efficiently by solving (12.33) and (12.34). To this end, just observe that, in the solution of the n continuous knapsack problems, if a Lagrangian profit $\hat{p}_{ij}$ happens to be nonpositive, the corresponding variable can be fixed to 0.

The Lagrangian dual problem (12.37) can now be seen as the determination of a matrix Λ by splitting each profit $2p_{ij}$ between the two objective function coefficients $\hat{p}_{ij}$ and $\hat{p}_{ji}$, so that the optimal solution of the relaxed problem is minimized. Caprara, Pisinger and Toth [69] prove, without loss of generality, that $\hat{p}_{ij} \geq 0$ for all $i, j \in N, j \neq i$.

A well-known result in Lagrangian relaxation (see e.g. Fisher [147]) states that the upper bound $z(L_2(\text{QKP}, \Lambda^*))$, where Λ^* is an optimal multiplier matrix, coincides with the optimal value of the LP-relaxation (12.39)–(12.43). However, the exact solution of this LP-relaxation would be computationally very expensive due to the large number of variables and constraints involved, although one could add the constraints to the model gradually, as described in Billionnet and Calmels [37].

Reformulation. For each Λ such that $\hat{p}_{ji} \geq 0$ for $i, j \in N, j \neq i$, the corresponding Lagrangian profit matrix $\hat{P}$ defines a (QKP) instance which is equivalent to the initial one, i.e., we have a *reformulation* of the original problem. The use of problem reformulations for the *quadratic assignment problem* was proposed by a few authors, see the paper by Carraresi and Malucelli [71] for a unified analysis of the various approaches.

Caprara, Pisinger and Toth used the reformulation associated with the best upper bound obtained at the root node throughout the branch-and-bound algorithm. This results in a quite tight bound which can be derived very efficiently.

12.2.6 Bounds from Lagrangian Decomposition

Two bounds based on Lagrangian decomposition have been presented for the quadratic knapsack problem. As described in Section 3.9 the idea of Lagrangian decomposition is to introduce some copy variables which are linked to the original variables through a number of equality constraints. Lagrangian relaxing the equality constraints, one is able to split the problem into a number of subproblems formulated in the disjoint sets of variables.

Michelon, Veuilleux. Michelon and Veilleux [347] consider the formulation

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in N} \sum_{j \in N} p_{ij} x_i x_j \\ \text{subject to} \quad & \sum_{j \in N} w_j y_j \leq c, \\ & x_j = y_j, \quad j \in N, \\ & x_j, y_j \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.47}$$

Lagrangian relaxing the constraints $x_j = y_j$ for $j \in N$ using multipliers $\lambda_j \in \mathbb{R}$ the problem $L_3(\text{QKP},\lambda)$ is obtained as

$$\begin{aligned} \text{maximize} & \sum_{i\in N}\sum_{j\in N} p_{ij}x_i x_j - \sum_{j\in N}\lambda_j(x_j - y_j) \\ \text{subject to} & \sum_{j\in N} w_j y_j \le c, \\ & x_j, y_j \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.48}$$

Defining *Lagrangian profits* $\tilde{p}_{ij}$ as

$$\tilde{p}_{ij} := \begin{cases} p_{ij} & \text{if } i \neq j, \\ p_{ij} - \lambda_j & \text{if } i = j, \end{cases} \tag{12.49}$$

problem $L_3(\text{QKP},\lambda)$ may be decomposed into two subproblems

$$\begin{aligned} \text{(QP)} \qquad & \text{maximize} \sum_{i\in N}\sum_{j\in N} \tilde{p}_{ij}x_i x_j \\ & \text{subject to } x_j, \in \{0,1\}, \quad j \in N, \end{aligned} \tag{12.50}$$

and

$$\begin{aligned} \text{maximize} & \sum_{i\in N} \lambda_j y_j \\ \text{subject to} & \sum_{j\in N} w_j y_j \le c, \\ & y_j \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.51}$$

The second subproblem (12.51) is an ordinary knapsack problem, which can be solved through the techniques described in Chapter 5. The first subproblem (QP) can be solved in a similar way as the model $L_1(\text{QKP},\lambda)$ described in Section 12.2.2. We construct a network $\mathcal{N} = (V,E,\bar{c})$ defined on the vertices $V = \{s,1,\ldots,n,t\}$ where s is the source and t is the sink. The edge set E is defined on $V \times V$, having the capacities

$$\begin{aligned} \bar{c}_{si}(\lambda) &= \max\left\{0, \textstyle\sum_{j\in N} p_{ij} - \lambda_i\right\}, \\ \bar{c}_{ij}(\lambda) &= p_{ij}, \\ \bar{c}_{it}(\lambda) &= \max\left\{0, \lambda_i - \textstyle\sum_{j\in N} p_{ij}\right\}. \end{aligned} \tag{12.52}$$

Assume that the value of the maximum flow from s to t in $\mathcal{N}$ is $\Psi(\mathcal{N})$.

Proposition 12.2.2 *The solution value $z(QP)$ to (12.50) is given by*

$$z(QP) := \sum_{i\in N} \bar{c}_{si} - \Psi(\mathcal{N})$$

Proof. As in the proof of Proposition 12.2.1 we define a cut $c(S,T)$ which separates the set of nodes V, into two subsets S and T. As before we assume that $s \in S$ and $t \in T$, and we use binary variables x_i to indicate whether node i is in set S. Since the maximum flow $\Psi(G)$ equals a minimum cut $c(S,T)$, the remaining reasoning follows the proof of Proposition 12.2.1. □

For any given matrix Λ of Lagrangian multipliers, we get the bound $U^1_{\text{MV}}(\Lambda)$. In order to get the tightest bound, the Lagrangian dual problem is solved

$$U^2_{\text{MV}} := \min_{\Lambda \in \mathbb{R}^{n \times n}} U^1_{\text{MV}}(\Lambda). \tag{12.53}$$

Using subgradient optimization, one gets the bound $\hat{U}^2_{\text{MV}}$ for a near-optimal matrix of Lagrangian multipliers $\hat{\Lambda}$. The bounds obviously satisfy $U^2_{\text{MV}} \leq \hat{U}^2_{\text{MV}} \leq U^1_{\text{MV}}(0)$, assuming that the subgradient optimization starts with $\Lambda = 0$.

Proposition 12.2.3 $U^2_{\text{MV}} \leq U_{\text{CHM}}$.

Proof. Let λ' be the optimal Lagrangian multiplier corresponding to the Lagrangian dual (12.11). Set $\Lambda' = (w_1\lambda', w_2\lambda', \ldots, w_n\lambda')$. Then for any feasible solution x to $L_1(\text{QKP}, \lambda')$ given by (12.3) all feasible solutions (x,y) to $L_3(\text{QKP}, \Lambda')$ given by (12.48) will satisfy

$$\begin{aligned} U_{\text{CHM}} - U^1_{\text{MV}}(\Lambda') &= \sum_{i \in N}\sum_{j \in N} p_{ij}x_ix_j - \lambda'\left(\sum_{j \in N} w_jx_j - c\right) \\ &\quad - \sum_{i \in N}\sum_{j \in N} p_{ij}x_ix_j + \sum_{j \in N} \lambda'_j(x_j - y_j) \\ &= \lambda'\left(c - \sum_{j \in N} w_jy_j\right) \geq 0, \end{aligned} \tag{12.54}$$

where the last inequality holds due to constraint (12.48) and the fact that $\lambda' \geq 0$. □

Billionnet, Faye and Soutif. The bound by Billionnet, Faye and Soutif [38] is based on a partitioning of N into m disjoint classes $N_1, \ldots, N_m$ satisfying $\cup_{k=1}^m N_k = N$. The main idea in the bound by Billionnet, Faye and Soutif is to use Lagrangian decomposition to split the problem into m independent subproblems. Each subproblem is solved by enumerating all decision variables in class N_k. For a fixed solution vector x_{N_k} the subproblem is an ordinary linear knapsack problem which may be solved in $O(n)$ time.

In order to partition the problem we notice that the objective function of (QKP) may be written

$$\sum_{i\in N}\sum_{j\in N} p_{ij}x_ix_j = \sum_{k=1}^{m}\sum_{i\in N_k}\left(\sum_{j\in N_k} p_{ij}x_ix_j + \sum_{j\in N\setminus N_k} p_{ij}x_ix_j\right) \tag{12.55}$$
$$= \sum_{k=1}^{m}\left(\sum_{i\in N_k}\sum_{j\in N_k} p_{ij}x_ix_j + \sum_{i\in N_k}\sum_{j\in N\setminus N_k} p_{ij}x_ix_j\right).$$

Using Lagrangian decomposition we introduce binary copy variables $y_j^k = x_j$ for $j \in N \setminus N_k$ and add k redundant capacity constraints. Moreover, we add copy constraints concerning the quadratic terms $x_ix_j = x_iy_j^k$, getting the following formulation. The function set(i) returns the set index of the class to which item i belongs, i.e. the index k for which $i \in N_k$. Since the sets N_k are disjoint, set(i) is well defined.

$$\begin{aligned}
\text{maximize } & \sum_{k=1}^{m}\left(\sum_{i\in N_k}\sum_{j\in N_k} p_{ij}x_ix_j + \sum_{i\in N_k}\sum_{j\in N\setminus N_k} p_{ij}x_iy_j^k\right) \\
\text{subject to } & x_j = y_j^k, && k = 1,\ldots,m, j \in N\setminus N_k, \\
& x_iy_j^{\text{set}(i)} = x_jy_i^{\text{set}(j)}, && i \in N, j \in N, \text{set}(i) \neq \text{set}(j), \\
& \sum_{i\in N_k} w_ix_i + \sum_{j\in N\setminus N_k} w_jy_j^k \leq c, && k = 1,\ldots,m, \\
& x_i \in \{0,1\}, && i \in N_k, k = 1,\ldots,m, \\
& y_j^k \in \{0,1\}, && k = 1,\ldots,m, j \in N\setminus N_k.
\end{aligned} \tag{12.56}$$

Lagrangian relaxing the two first constraints using multipliers $\Lambda = \{\lambda_j^k\}$, $k = 1,\ldots,m, j \in N \setminus N_k$ respectively $M = \{\mu_{ij}\}$, $i \in N, j \in N, \text{set}(i) \neq \text{set}(j)$ we get

$$\begin{aligned}
\text{maximize } & \sum_{k=1}^{m}\sum_{i\in N_k}\sum_{j\in N_k} p_{ij}x_ix_j + \sum_{k=1}^{m}\sum_{i\in N_k}\sum_{j\in N\setminus N_k} p_{ij}x_iy_j^k + \\
& \sum_{k=1}^{m}\sum_{j\in N\setminus N_k}\lambda_j^k(x_j - y_j^k) + \sum_{i\in N}\sum_{\substack{j\in N \\ \text{set}(j)\neq\text{set}(i)}}\mu_{ij}\left(x_iy_j^{\text{set}(i)} - x_jy_i^{\text{set}(j)}\right) \\
\text{subject to } & \sum_{j\in N_k} w_ix_i + \sum_{j\in N\setminus N_k} w_jy_j^k \leq c, && k = 1,\ldots,m, \\
& x_i \in \{0,1\}, && i \in N_k, k = 1,\ldots,m \\
& y_j^k \in \{0,1\}, && k = 1,\ldots,m, j \in N\setminus N_k.
\end{aligned} \tag{12.57}$$

Since

$$\sum_{i\in N}\sum_{\substack{j\in N\\ \mathrm{set}(j)\neq \mathrm{set}(i)}} \mu_{ij}\left(x_i y_j^{\mathrm{set}(i)} - x_j y_i^{\mathrm{set}(j)}\right) =$$
$$\sum_{k=1}^{m}\sum_{i\in N_k}\sum_{j\in N\setminus N_k} \mu_{ij}x_i y_j^k - \sum_{k=1}^{m}\sum_{i\in N_k}\sum_{j\in N\setminus N_k} \mu_{ji}x_i y_j^k,$$

and

$$\sum_{k=1}^{m}\sum_{j\in N\setminus N_k}\lambda_j^k x_j = \sum_{i\in N}\left(\sum_{h\neq \mathrm{set}(i)}\lambda_i^h\right)x_i = \sum_{k=1}^{m}\sum_{i\in N_k}\left(\sum_{h\neq k}\lambda_i^h\right)x_i,$$

we can split (12.57) into m independent problems $L_k(N_k,\Lambda,M), k=1,\ldots,m$ of the form

$$\begin{aligned}
\text{maximize } & \sum_{i\in N_k}\sum_{j\in N_k} p_{ij}x_i x_j + \sum_{i\in N_k}\sum_{j\in N\setminus N_k} p_{ij}x_i y_j^k \\
& + \sum_{i\in N_k}\left(\sum_{h\neq k}\lambda_i^h\right)x_i - \sum_{j\in N\setminus N_k}\lambda_j^k y_j^k \\
& + \sum_{i\in N_k}\sum_{j\in N\setminus N_k}(\mu_{ij}-\mu_{ji})x_i y_j^k \qquad (12.58)\\
\text{subject to } & \sum_{i\in N_k} w_i x_i + \sum_{j\in N\setminus N_k} w_j y_j^k \le c, \\
& x_i\in\{0,1\}, \qquad i\in N_k, k=1,\ldots,m, \\
& y_j^k\in\{0,1\}, \qquad j\in N\setminus N_k.
\end{aligned}$$

Rearranging the terms we get

$$\begin{aligned}
\text{maximize } & \sum_{i\in N_k}\sum_{j\in N_k} p_{ij}x_i x_j + \sum_{i\in N_k}\left(\sum_{h\neq k}\lambda_i^h\right)x_i \\
& + \sum_{j\in N\setminus N_k}\left(\sum_{i\in N_k} p_{ij}x_i\right)y_j^k - \sum_{j\in N\setminus N_k}\lambda_j^k y_j^k \\
& + \sum_{j\in N\setminus N_k}\left(\sum_{i\in N_k}(\mu_{ij}-\mu_{ji})x_i\right)y_j^k \qquad (12.59)\\
\text{subject to } & \sum_{i\in N_k} w_i x_i + \sum_{j\in N\setminus N_k} w_j y_j^k \le c, \\
& x_i\in\{0,1\}, \qquad i\in N_k, \\
& y_j^k\in\{0,1\}, \qquad j\in N\setminus N_k.
\end{aligned}$$

Assuming that the sets N_k are small we may enumerate all solutions of N_k in problem $L_k(N_k,\Lambda,M)$. For a fixed value of $x_i, i\in N_k$ the problem can be recognized as a

knapsack problem defined in the variables y_j^k by setting

$$\tilde{p}_j := \sum_{i \in N_k} p_{ij} x_i - \lambda_j^k + \sum_{i \in N_k} (\mu_{ij} - \mu_{ji}) x_i,$$

hence getting $L_k(N_k, \Lambda, M)$ in the form

$$\begin{aligned} &\text{maximize} \quad \text{const} + \sum_{j \in N \setminus N_k} \tilde{p}_j y_j^k \\ &\text{subject to} \sum_{j \in N \setminus N_k} w_j y_j^k \leq c - \sum_{i \in N_k} w_i x_i, \\ &\qquad y_j^k \in \{0,1\}, \quad j \in N. \end{aligned} \tag{12.60}$$

The objective function may hence be written as

$$U_{\text{BFS}}^1(\Lambda, M) := \sum_{k=1}^{m} L_k(N_k, \Lambda, M). \tag{12.61}$$

In order to make the computation of $U_{\text{BFS}}^1(\Lambda, M)$ faster we solve the continuous relaxation of (12.60) which can be done in $O(n)$ time. The tightest bound U_{BFS}^2 is now found as a solution to the Lagrangian dual problem

$$U_{\text{BFS}}^2 := \min_{\Lambda, M} U_{\text{BFS}}^1(\Lambda, M). \tag{12.62}$$

A sub-optimal choice of Lagrangian multipliers Λ and M can be found through subgradient optimization as described in Section 3.8 leading to the bound $\hat{U}_{\text{BFS}}^2$ with $U_{\text{BFS}}^2 \leq \hat{U}_{\text{BFS}}^2 \leq U_{\text{BFS}}^1(0,0)$.

The time complexity of the bound for given values of Λ, M is derived as follows. If we assume that the set N was partitioned into m sets N_k of equal size n/m then the time consumption for solving problem $L_k(N_k, \Lambda, M)$ for a given choice of λ, μ is $O(2^{n/m} n)$. As we have to solve m subproblems the total time consumption is $O(2^{n/m} mn)$. Billionnet, Faye and Soutif [38] notice that the larger the size n/m the better bounds are observed — however at the cost of an exponentially increasing time consumption. The time complexity should be multiplied by the number of iterations used for iterating the Lagrangian multipliers in the bound $\hat{U}_{\text{BFS}}^2$. In their computational experiments, Billionnet, Faye and Soutif used relatively small sets of size $n/m = 5$ to keep the computational times at a reasonable level.

12.2.7 Bounds from Semidefinite Relaxation

Helmberg, Rendl and Weismantel [219, 220] and Helmberg [218] proposed a number of upper bounds for (QKP) based on semidefinite programming. These bounds

are valid for a more general version of the problem where the profit matrix P may contain negative entries. For reasons of completeness, the bounds are presented in the sequel, although it is outside the scope of the present book to give a deeper introduction to semidefinite programming. Instead we refer e.g. to Wolkowicz, Saigal and Vandenberghe [487].

Let us first introduce some notation. We will write the solution variables and the weights as column vectors $x = (x_1, \ldots, x_n)^T$, $w = (w_1, \ldots, w_n)^T$. The set of real $m \times n$ matrices is denoted by $M_{m,n}$. The *inner product between matrices* $A, B \in M_{m,n}$ is defined as

$$\langle A, B \rangle := \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} b_{ij}. \tag{12.63}$$

The *vector product* of two vectors $a, b \in \mathbb{R}^n$ is

$$ab^T = \begin{pmatrix} a_1 b_1 & \cdots & a_1 b_n \\ \vdots & & \vdots \\ a_n b_1 & \cdots & a_n b_n \end{pmatrix}. \tag{12.64}$$

The *vector of diagonal elements* of a square matrix $A \in M_{n,n}$ will be denoted $\mathrm{diag}(A)$ and is defined as follows

$$\mathrm{diag}(A) := \mathrm{diag} \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} \\ \vdots \\ a_{nn} \end{pmatrix}. \tag{12.65}$$

The somehow reverse operation $\mathrm{Diag}(a)$ takes a vector $a \in \mathbb{R}^n$ and converts it into a *diagonal matrix.*

$$\mathrm{Diag}(a) = \mathrm{Diag}(a_1, \ldots, a_n) = \begin{pmatrix} a_1 & 0 & \cdots & 0 \\ 0 & a_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & a_n \end{pmatrix}. \tag{12.66}$$

The *rank of a matrix* A will be written as $\mathrm{rank}(A)$. It is defined as the number of linearly independent columns of A. A matrix $A \in M_{n,n}$ is said to be *positive semidefinite* if we have for all vectors $y \in \mathbb{R}^n$

$$y^T A y \geq 0. \tag{12.67}$$

We will use the notation $A \succeq 0$ to indicate that A is symmetric and positive semidefinite. A *semidefinite optimization problem* in standard form is written as

$$\begin{aligned} &\text{maximize } \langle P, X \rangle \\ &\text{subject to } \langle A_1, X \rangle = b_1, \end{aligned} \tag{12.68}$$

$$\begin{aligned}&\vdots\\&\langle A_m, X\rangle = b_m,\\&X \succeq 0\end{aligned}$$

where $X = (X_{ij}) \in M_{n,n}$, $P \in M_{n,n}$, $A_i \in M_{n,n}$, and $b_i \in \mathbb{R}$.

A problem defined in several matrices $X_1, \ldots, X_k$ can be expressed in the above form by observing that

$$X_1 \succeq 0, X_2 \succeq 0, \ldots, X_k \succeq 0 \quad \Leftrightarrow \quad \begin{pmatrix} X_1 & 0 & \cdots & 0 \\ 0 & X_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & X_k \end{pmatrix} \succeq 0. \tag{12.69}$$

The observation makes it easy to formulate LP-problems as semidefinite optimization problems. Nonnegativity constraints may be expressed by formulating each constraint $x_i \geq 0$ as a 1×1 positive semidefinite matrix $X_i = (x_i)$. In this way constraint (12.67) is equivalent to $xy^2 \geq 0$ and further $x \geq 0$. Linear constraints $\sum_{j=1}^{n} a_j x_j \leq b$ are easily expressed as an inner product $\langle A, X\rangle = b$ with $A = \mathrm{Diag}(a)$ and the additional constraints $X_{ij} = 0$ for $i \neq j$ which ensures that X is a diagonal matrix. Inequalities are transformed to equations by adding nonnegative slack variables.

Grötschel, Lovász and Schrijver [199] proved that a semidefinite optimization problem in the form (12.68) can be solved in polynomial time measured in the input size and accuracy, hence making *semidefinite relaxations* an attractive tool for deriving upper bounds in combinatorial optimization. For a concrete implementation of semidefinite optimizers see e.g. Sturm [446], Toh, Tutuncu and Todd [460], Fujisawa, Kojima and Nakata [158].

In order to formulate a number of bounds for the (QKP) we will use the following proposition:

Proposition 12.2.4 *The following two properties are equivalent for a matrix $X \in M_{n,n}$:*

1. $X \succeq 0$ and $\mathrm{rank}(X) = 1$.

2. $X = xx^T$ for some vector $x \in \mathbb{R}^n$.

Proof. See e.g. Helmberg [218]. □

The consequence of the above proposition is that we may write the objective function of (QKP) as

$$x^T P x = \sum_{i \in N} \sum_{j \in N} p_{ij} x_i x_j = \langle P, X\rangle. \tag{12.70}$$

This leads to the following formulation of (QKP):

$$\begin{aligned} &\text{maximize } \langle P, X\rangle \\ &\text{subject to } \langle \mathrm{Diag}(w), X\rangle \le c, \\ &\qquad X \succeq 0, \\ &\qquad \mathrm{rank}(X) = 1, \\ &\qquad X_{ii} \in \{0,1\}. \end{aligned} \tag{12.71}$$

By dropping the $\mathrm{rank}(X) = 1$ constraint, and relaxing the last constraint to $0 \le X_{ii} \le 1$, we get a semidefinite relaxation which gives us an upper bound U^0_{HRW} on (QKP). According to Helmberg, Rendl and Weismantel [220] this bound is of poor quality.

To reach tighter bounds, we observe that we may reformulate the last three constraints in formulation (12.71).

Proposition 12.2.5 *If* $X \succeq 0$ *and* $\mathrm{rank}(X) = 1$ *and* $X_{ii} \in \{0,1\}$ *then also*

$$X - \mathrm{diag}(X)\,\mathrm{diag}(X)^T \succeq 0. \tag{12.72}$$

Proof. Due to Proposition 12.2.4 we can write $X = xx^T$. For any vector $v \in \mathbb{R}^n$ we have that the matrix $(x+v)(x+v)^T \succeq 0$. Hence by multiplication

$$xx^T + vx^T + xv^T + vv^T \succeq 0, \qquad v \in \mathbb{R}^n.$$

Using the fact that $\mathrm{diag}(xx^T) = x$ when $x \in \{0,1\}^n$ we get

$$\begin{aligned} X + v\,\mathrm{diag}(X)^T + \mathrm{diag}(X)v^T + vv^T \succeq 0, \quad v \in \mathbb{R}^n \quad &\Leftrightarrow \\ X + (v + \mathrm{diag}(X))(v + \mathrm{diag}(X))^T - \mathrm{diag}(X)\,\mathrm{diag}(X)^T \succeq 0, \quad v \in \mathbb{R}^n. \end{aligned}$$

Choosing $v = -\mathrm{diag}(X)$ we get the stated. □

To express (12.72) as a semidefinite optimization problem we note that

Proposition 12.2.6

$$X - \mathrm{diag}(X)\,\mathrm{diag}(X)^T \succeq 0 \qquad \Leftrightarrow \qquad \overline{X} \succeq 0 \tag{12.73}$$

where

$$\overline{X} := \begin{pmatrix} 1 & \mathrm{diag}(X)^T \\ \mathrm{diag}(X) & X \end{pmatrix}. \tag{12.74}$$

Proof. Define the regular matrix B as

$$B = \begin{pmatrix} 1 & -\mathrm{diag}(X)^T \\ 0 & I \end{pmatrix},$$

and observe that

$$B^T\overline{X}B = \begin{pmatrix} 1 & 0 \\ -\operatorname{diag}(X) & I \end{pmatrix} \begin{pmatrix} 1 & \operatorname{diag}(X)^T \\ \operatorname{diag}(X) & X \end{pmatrix} \begin{pmatrix} 1 & -\operatorname{diag}(X)^T \\ 0 & I \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ 0 & X - \operatorname{diag}(X)\operatorname{diag}(X)^T \end{pmatrix}. \tag{12.75}$$

Due to the fact that

$$y^T\overline{X}y = y^T(B^{-1})^T B^T\overline{X}BB^{-1}y = \overline{y}^T B^T\overline{X}B\overline{y}, \tag{12.76}$$

with $\overline{y} = B^{-1}y$, we have that $\overline{X}$ satisfies property (12.67) for a given y if and only if $B^T\overline{X}B$ satisfies the property. Hence $\overline{X} \succeq 0$, if and only if $B^T\overline{X}B \succeq 0$. Observation (12.69) applied to (12.75) now gives the stated. □

Based on the two above propositions, we may relax (QKP) to (QKP') of the form

$$\begin{aligned} &\text{maximize } \langle P, X\rangle \\ &\text{subject to } \langle \operatorname{Diag}(w), X\rangle \le c, \\ &\qquad X - \operatorname{diag}(X)\operatorname{diag}(X)^T \succeq 0, \\ &\qquad \operatorname{rank}(X) = 1, \\ &\qquad X_{ii} \in \{0,1\}. \end{aligned} \tag{12.77}$$

The bound U^1_{HRW} is obtained by dropping the $\operatorname{rank}(X) = 1$ constraint and relaxing the last constraint to $0 \le X_{ii} \le 1$ getting the model

$$\begin{aligned} &\text{maximize } \langle P, X\rangle \\ &\text{subject to } \langle \operatorname{Diag}(w), X\rangle \le c, \\ &\qquad X - \operatorname{diag}(X)\operatorname{diag}(X)^T \succeq 0, \\ &\qquad X_{ii} \le 1. \end{aligned} \tag{12.78}$$

The next bound U^2_{HRW} is based on the fact that $w^T x = x^T w$, hence $w^T x \le c$ implies that $w^T x x^T w \le c^2$. By rewriting $w^T x x^T w$ as $\langle ww^T, xx^T\rangle$ and relaxing xx^T to X we get the relaxation

$$\begin{aligned} &\text{maximize } \langle P, X\rangle \\ &\text{subject to } \langle ww^T, X\rangle \le c^2, \\ &\qquad X - \operatorname{diag}(X)\operatorname{diag}(X)^T \succeq 0. \end{aligned}$$

The third semidefinite relaxation is based on the observation that $w^T x \le c$ can be multiplied by the real number $w^T x$ on both sides gives the constraint $(w^T x)^2 \le w^T x c$, leading to the inequality

$$\begin{aligned} 0 \le w^T x(c - x^T w) &= w^T x \begin{pmatrix} 1, x^T \end{pmatrix} \begin{pmatrix} c \\ -w \end{pmatrix} \\ &= \begin{pmatrix} 0, w^T \end{pmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix} \begin{pmatrix} 1, x^T \end{pmatrix} \begin{pmatrix} c \\ -w \end{pmatrix}. \end{aligned} \tag{12.79}$$

Setting $X' = \begin{pmatrix} 1 \\ x \end{pmatrix} \begin{pmatrix} 1, x^T \end{pmatrix} = x'x'^T$ the right hand side expression in (12.79) can be written

$$\left\langle \begin{pmatrix} c \\ -w \end{pmatrix} \begin{pmatrix} 0, w^T \end{pmatrix}, X' \right\rangle. \tag{12.80}$$

This leads to the following relaxation:

$$\begin{aligned} &\text{maximize } \langle P, X \rangle \\ &\text{subject to } \left\langle \begin{pmatrix} c \\ -w \end{pmatrix} \begin{pmatrix} 0, w^T \end{pmatrix}, X' \right\rangle \geq 0, \\ &\qquad X - \operatorname{diag}(X) \operatorname{diag}(X)^T \succeq 0. \end{aligned} \tag{12.81}$$

Solving the above problem gives bound U^3_{HRW}.

The last relaxation is obtained by multiplying the capacity constraint with each of the variables x_i for $i \in N$ getting $x_i w^T x \leq x_i c$. By writing the vector product explicitly we get the sum

$$\sum_{j \in N} w_j x_i x_j \leq x_i c. \tag{12.82}$$

By introducing in (12.82) X_{ij} for $x_i x_j$ and X_{ii} for x_i we get

$$\sum_{j \in N} w_j X_{ij} \leq X_{ii} c, \tag{12.83}$$

which leads to the following relaxation:

$$\begin{aligned} &\text{maximize } \langle P, X \rangle \\ &\text{subject to } \sum_{j \in N} w_j X_{ij} - X_{ii} c \leq 0, \quad \text{for } i \in N, \\ &\qquad X - \operatorname{diag}(X) \operatorname{diag}(X)^T \succeq 0, \end{aligned} \tag{12.84}$$

giving the bound U^4_{HRW}.

Helmberg, Rendl and Weismantel [220] together with Bauvin and Goemans [29] prove that $U^1_{\text{HRW}} \geq U^2_{\text{HRW}} \geq U^3_{\text{HRW}} \geq U^4_{\text{HRW}}$ hence showing that the formulation (12.84) is to be preferred. More precisely they showed:

Proposition 12.2.7 *Let $\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3, \mathcal{X}_4$ be the feasible sets corresponding to bounds $U^1_{\text{HRW}}, U^2_{\text{HRW}}, U^3_{\text{HRW}}$ and U^4_{HRW}. Then we have the relation $\mathcal{X}_1 \supseteq \mathcal{X}_2 \supseteq \mathcal{X}_3 \supseteq \mathcal{X}_4$.*

Proof. To prove that $\mathcal{X}_1 \supseteq \mathcal{X}_2$ assume that $X \in \mathcal{X}_2$. Introducing the positive semidefinite matrix $Z = X - \operatorname{diag}(X) \operatorname{diag}(X)^T$ we have

$$c^2 \geq w^T X w = wZw + (w^T \operatorname{diag}(X))^2. \tag{12.85}$$

Since $Z \succeq 0$ by the feasibility of X, it follows that $(w^T \operatorname{diag}(X))^2 \leq c^2$, which proves $X \in \mathcal{X}_1$.

Next, let $X \in \mathcal{X}_3$. Using the same matrix $Z = X - \operatorname{diag}(X)\operatorname{diag}(X)^T \succeq 0$ we see that

$$\begin{aligned} 0 &\leq \left\langle \begin{pmatrix} c \\ -w \end{pmatrix} \begin{pmatrix} 0 & w^T \end{pmatrix}, X' \right\rangle \\ &= cw^T \operatorname{diag}(X) - w^T X w \\ &= cw^T \operatorname{diag}(X) - w^T Z w - w^T \operatorname{diag}(X)\operatorname{diag}(X)^T w \\ &= \left(c - w^T \operatorname{diag}(X)\right) w^T \operatorname{diag}(X) - w^T Z w. \end{aligned} \tag{12.86}$$

Since $-w^T Z w \leq 0$ we have $c \geq w^T \operatorname{diag}(X)$ and hence $c^2 \geq w^T X w$ in the first row of the equation. This shows that $X \in \mathcal{X}_2$.

Finally, assume that $X \in \mathcal{X}_4$, and multiply each x_i representation of (12.84) by $w_i \geq 0$. Summing all these inequalities over i we get

$$\sum_{i \in N} \sum_{j \in N} w_i w_j x_{ij} = \langle ww^T, X \rangle \leq \sum_{i \in N} c w_i x_{ii}, \tag{12.87}$$

which gives exactly the inequality of (12.81), hence $X \in \mathcal{X}_4$. □

12.3 Variable Reduction

The size of a (QKP) instance may in many cases be considerably reduced by using some reduction rules from the classical knapsack problem as described in Section 3.2. Assume that an incumbent solution of value z^ℓ has been determined by some initial heuristic. Let U_j^1 be an upper bound on the (QKP) obtained by imposing the additional constraint $x_j = 1$. If $U_j^1 \leq z^\ell$ then we can fix x_j at 0. Similarly, if U_j^0 is an upper bound on the (QKP) obtained by imposing the additional constraint $x_j = 0$ and $U_j^0 \leq z^\ell$, we can fix x_j at 1. Whenever a variable x_j is fixed at some value, we remove the corresponding row and column. Moreover, if it is fixed at 1, we also increase diagonal entry p_{ii} by $p_{ij} + p_{ji}$, for $i \in N \setminus \{j\}$, and decrease c by w_j.

For the reduction, Caprara, Pisinger and Toth [69] used the bound U_{CPT}^1 which can be determined in $O(n^2)$ time for each j by solving the Lagrangian relaxed problem for a fixed set of Λ values corresponding to the solution of (12.37) at the root node of the branch-and-bound tree. If the reduction procedure fixes at least one variable, subgradient optimization may be applied to the now reduced problem, followed by a new reduction. In worst-case this approach runs in $O(n^4)$ although in practice only a very limited number of iterations are needed for the latter part.

Hammer and Rader [210] used the bound U_{CHM} based on Lagrangian relaxation of the capacity constraint as described in Section 12.2.2. In addition, they used some order relations which may be used to fix variables at their proper value inside a branch-and-bound algorithm: Assume that two items i, j satisfy that $w_i \geq w_j$, and consider the so-called "second derivative" Δ_{ij} given by

$$\Delta_{ij} = p_{ii} - p_{jj} + \sum_{k \in N \setminus \{i,j\}} (p_{ik} - p_{jk}) x_k \,. \tag{12.88}$$

If $\Delta_{ij} \leq 0$ at some optimal solution x, then there exists an optimal solution x^* so that $x_i^* \leq x_j^*$. Hence, whenever we branch at $x_i^* = 1$ we may immediately fix $x_j^* = 1$, and if we branch at $x_j^* = 0$ we may fix $x_i^* = 0$. We refer to [210] for additional details.

12.4 Branch-and-Bound

Several branch-and-bound algorithms for (QKP) have been presented in the literature, the main distinction being the upper bounds used.

Gallo, Hammer and Simeone [160] developed the first branch-and-bound algorithm using the bounds U^1_{GHS} to U^4_{GHS} based on upper planes. The branching variable was selected by solving (12.12) and letting F be the set of variables for which $x_j = 1$ in the present solution. Branching is then performed on the variable $j \in F$ which maximizes $\frac{\pi_j}{w_j}$.

Chaillou, Hansen and Mahieu [75] as well as Hammer and Rader [210] used the bound U_{CHM} from Lagrangian relaxation of the capacity constraint. The branching strategy by Chaillou, Hansen and Mahieu was based on choosing the variable which results in the smallest reduction in the bound when changing the variable to its complement. In this derivation of upper bounds an approximation of U_{CHM} is used based on the same value of λ, as obtained in Step 3 of Figure 12.2. Hammer and Rader followed the strategy to do as much analysis as possible at each branching node in order to get a limited search tree. Hence, a three-step procedure was developed, making use of *constraint pairing* (see also Section 9.2), fixation of variables by Lagrangian techniques, and order relations (see Section 12.3). Branching is performed on the variable which makes it possible to fix most possible variables at their optimal value.

Billionnet and Calmels [37] applied the bound U^2_{BC} using a branch-and-cut approach for gradually building up the model. When deciding which variable to branch at, they consider the solution vector x^* of the model (12.24) to (12.31). Branching is then performed on the variable i which maximizes the quantity $|x_i^* - \frac{1}{2}|$.

Helmberg, Rendl and Weismantel [219] used the bounds U^0_{HRW} to U^4_{HRW} in their computational study. The problem is solved through cutting plane techniques, where additional constraints are used to tighten the formulation. In several cases, the upper and lower bound coincide at the root node, hence solving the problem to optimality without any branching.

Caprara, Pisinger and Toth [69] used the bound $\hat{U}^2_{\text{CPT}}$, in their depth-first branch-and-bound algorithm, although the Lagrangian dual (12.37) was only solved at the root node, and the given Lagrangian profits $(\hat{p}_{ij})$ were used in deriving U^{1c}_{CPT} in subsequent

nodes. The branching order is determined in advance at the root node, based on the values

$$\pi'_i := p_{ii} + \max \left\{ \sum_{j \in N \setminus \{i\}} \hat{p}_{ji} \bar{x}_j \;\middle|\; \begin{array}{l} \sum_{j \in N \setminus \{i\}} w_j \bar{x}_j \leq c - w_i, \\ 0 \leq \bar{x}_j \leq 1, j \in N \setminus \{i\} \end{array} \right\}. \tag{12.89}$$

This can be recognized as the upper planes (12.33), however with rows and columns interchanged. The motivation for using π'_i instead of π_j^{5*} is that the latter profits are "flattened" by the subgradient optimization procedure, hence leaving no information for defining the branching order. The variables are reordered according to decreasing values of π'_i, and each branching node branches on the variable with the smallest index among the free variables.

It should finally be mentioned that no branch-and-bound algorithm has been presented which makes use of the bound U^2_{BFS} by Billionnet, Faye and Soutif [38]. In their computational study, the authors demonstrate the relative strength of the bound, although it is quite time consuming to derive.

12.5 The Algorithm by Caprara, Pisinger and Toth

We will now go into details with the exact algorithm QuadKnap by Caprara, Pisinger and Toth [69] as it has been able to solve some of the largest instances in the literature.

The main steps of the algorithm can be outlined as follows:

- Subgradient optimization with an embedded heuristic procedure is used to define tight upper and lower bounds, as well as a convenient problem reformulation.
- A reduction algorithm as described in Section 12.3 is used to fix variables at their optimal value. If the algorithm succeeds in fixing some variables, the above subgradient optimization step is repeated.
- Variables are reordered according to decreasing values of π'_i as defined by (12.89).
- A depth-first branch-and-bound algorithm is used for the final solution as described in the last part of Section 12.4.

In the following, N is the set of variables that were not fixed by the reduction procedure. Moreover, $\hat{\Lambda}$ is the matrix of Lagrangian multipliers associated with the best upper bound $\hat{U}^2_{\text{CPT}}$ found by the subgradient procedure, $\hat{P} = (\hat{p}_{ij})$ is the corresponding Lagrangian profit matrix, (p_{jj}) is the diagonal profit vector modified according to the variables fixed at 1 by the reduction, and (π_j^{5*}) contains the optimal objective function values of (LKP)'s (12.33) associated with matrix $\hat{P}$ rather than P.

In order to speed up the search, an auxiliary vector $\underline{w}$ is used to store the minimum succeeding weights, defined by

$$\underline{w}_i := \min_{j \geq i} w_i \quad \text{for } i \in N. \tag{12.90}$$

Obviously, whenever the branching mechanism has fixed variables x_j to $\bar{x}_j$, $j = 1,\ldots,i-1$, so that $\sum_{j=1}^{i-1} w_j \bar{x}_j + \underline{w}_i > c$, the algorithm can backtrack, since no other variable x_j, $j \geq i$, can be set to one.

The branching scheme can easily be described in a recursive way. Assuming that variables x_j, $j = 1,\ldots,i-1$, have been fixed at $\bar{x}_j$, we have the profit and weight sums

$$\Pi := \sum_{j=1}^{i-1} \sum_{k=1}^{i-1} p_{jk} \bar{x}_j \bar{x}_k, \qquad \Omega := \sum_{j=1}^{i-1} w_j \bar{x}_j. \tag{12.91}$$

The next variable x_i can either be set to 1 or to 0. In the first case the diagonal elements for $j > i$ are updated by setting $p_{jj} := p_{jj} + p_{ji} + p_{ij}$. In the second case, no updating must be performed. In both cases, the split items associated with (LKP)'s (12.33) are recomputed. By using parametric techniques described below, this upper bound can be computed in linear expected time.

As anticipated, subgradient optimization is applied only at the root node, whereas for the rest of the branch-and-bound algorithm we work on the (QKP) instance defined by the Lagrangian profit matrix $\hat{P}$. In particular, at each node an upper bound is derived by solving problem (12.45) subject to (12.40)–(12.42). The relaxed problem is defined on the free items, with capacity replaced by $c - \Omega$, and by adding Π to the optimal solution obtained. We backtrack if the upper bound U does not exceed the incumbent solution value z^ℓ. This leads to the recursive algorithm in Figure 12.4, which is initially called as QuadBranch(0,0,1), after the processing of the root node.

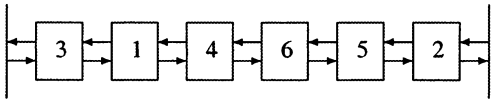
Deriving Upper Bounds in Linear Expected Time

The upper bound computation is the most time-consuming operation at any node of the branching tree. Hence Caprara, Pisinger and Toth used parametric techniques to ensure a linear expected time complexity. The key observation is that all the $n-1$ items of each (LKP) (12.33) problem are present at the root node, while some of them are removed during the branching. This means that it is only necessary to order the items at the root node according to decreasing efficiencies, and then use a double linked list to store the items which are still active, i.e., those whose variable has not been fixed by branching. Figure 12.5 shows the double linked list which is implemented by storing, for each item i, the sequence number r_i, the predecessor a_i and the successor b_i, according to the sorting. The pointers to the list elements corresponding to each item are stored in an array, so that we can directly access each item in the list. When branching on a variable x_i in the branch-and-bound algorithm,

Algorithm QuadBranch(Π, Ω, i)

```
if Π > z^ℓ then z^ℓ := Π, x* := x
if i ≤ n and Ω + w_i ≤ c then
  derive upper bound U^1c_CPT(Λ̂)
  if U^1c_CPT(Λ̂) > z^ℓ then
    for j := i+1 to n do
      Dequeue item i from the double-linked list corresponding to problem (12.33)
    branch on x_i = 1
    for j := i+1 to n do
      find the new split item s in each problem (12.33)
    for j := i+1 to n do p_jj := p_jj + p_ji + p_ij
    set x_i := 1
    call QuadBranch(Π + p_ii, Ω + w_i, i+1)
    for j := i+1 to n do p_jj := p_jj − p_ji − p_ij
    branch on x_i = 0
    for j := i+1 to n do
      find the new split item s in each problem (12.33)
    set x_i := 0
    call QuadBranch(Π, Ω, i+1)
    for j := i+1 to n do
      Enqueue item i in the double-linked list corresponding to problem (12.33)
  end if
end if
```

Fig. 12.4. The algorithm QuadBranch is a straightforward depth-first branch-and-bound algorithm making use of the bounds $U^{1c}_{\text{CPT}}(\hat{\Lambda})$ which can be calculated in expected $O(n)$ time for each branching node.



i	r_i	$\hat{p}_{ij}$	w_i	a_i	b_i
1	2	20	4	3	4
2	6	14	7	5	0
3	1	30	5	0	1
4	3	16	4	1	6
5	5	10	5	6	2
6	4	18	6	4	5

Fig. 12.5. Illustration of the double-linked list which is used to speed up solution of linear knapsack problems. For each problem (12.33) we maintain a double-linked list of the active items ordered by decreasing efficiencies, and an array of pointers to each element in the list.

we remove it from the double-linked list, using an ordinary Dequeue operation (see Cormen et al. [92, Section 10.2]). Upon backtracking, we insert it into the double-linked list, using the Dequeue operation.

Let $K := \{k \in N | k \text{ unfixed}\}$, $L := \{k \in N | k \text{ fixed at } 1\}$ and $c' := c - w_j - \sum_{k \in L} w_k$, and consider problem (12.33) associated with an unfixed item j, where N and $c - w_j$ are replaced by K and c', respectively. For this problem we maintain the current split item position s, the sums $w' = \sum_{k \in K, r_k < s} w_k$ and $p' = \sum_{k \in K, r_k < s} \hat{p}_{kj}$. During the branching process we must ensure that the split item position s is defined by

$$\sum_{k\in K, r_k<s} w_k \le c' < \sum_{k\in K, r_k\le s} w_k, \tag{12.92}$$

since then the objective value of (12.33) is found as $\overline{p} = p' + (c' - w')\hat{p}_{tj}/w_t$ where t is chosen such that $r_t = s$. Each iteration of the branch-and-bound algorithm may fix a variable x_i at 0 or 1, meaning that we must update s, p' and w' by one of the following steps:

1. $x_i = 1$, $r_i < s$: Both c' and w' are decreased by w_i while p' is decreased by $\hat{p}_{ij}$. The split item position s is unchanged.
2. $x_i = 1$, $r_i \ge s$: The capacity c' is decreased by w_i and we use linear search from s to the left in order to find the new value of s. Thus while $w' > c'$, repeatedly decrease p', w' by $\hat{p}_{tj}, w_t$, where t is such that $r_t = s - 1$, and decrease s by 1.
3. $x_i = 0$, $r_i > s$: All variables c', w', p' and s are unchanged.
4. $x_i = 0$, $r_i \le s$: If $r_i < s$ then the sums p', w' are decreased by $\hat{p}_{ij}$ and w_i, otherwise s is incremented by 1. Linear search from s to the right is used to find the new value of s. While $w' + w_t \le c'$, increase p', w' by $\hat{p}_{tj}, w_t$, where s is such that $r_t = s$, and increment s by 1.

Since the item i may be accessed and dequeued/enqueued in constant time, both Steps 1 and 3 may be performed in constant time. Steps 2 and 4 involve a linear search which in worst-case may demand $O(n)$ time, but on average only demands a constant number of operations under the assumption that each weight w_i is an independent random number in $[1, R]$. This is shown in the following two lemmas by Caprara, Pisinger and Toth [69]:

Lemma 12.5.1 *The expected number of iterations in the forward/backward search of Steps 2 and 4 are bounded by a constant.*

Proof. For every item i fixed at $x_i = 1$ or $x_i = 0$ we must search backwards or forward from the split item until the weight sum of the items passed is not smaller than w_i. Under the above stochastic assumption the statement follows from Lemma 1 in [69]. □

The following proposition, derived from Lemma 12.5.1, expresses the main feature of the branch-and-bound algorithm.

Lemma 12.5.2 *Every node of the branching tree is processed in $O(n)$ expected time.*

Proof. In a forward step of the branch-and-bound algorithm, from Lemma 12.5.1, each problem (12.33) is solved in constant expected time, and thus relaxation (12.45) subject to (12.40)–(12.42) is solved in linear expected time. Backtracking is performed in a similar way as forward steps, by enqueuing an item. Finally, if the variables s, p', w', c' are stored as part of each branching node, no additional computation is needed. □

12.6 Heuristics

We may roughly divide the heuristics for (QKP) into two classes: The *primal*, which maintain feasibility throughout the construction, and the *dual* which start from an infeasible solution and strive towards a feasible solution. Gallo, Hammer and Simeone [160] presented a family of primal heuristics corresponding to the bounds based on upper planes. Solving (12.12) immediately gives a feasible solution to (QKP). If the continuous relaxation of (12.12) is solved, one may obtain a feasible solution by truncating the fractional variables. Gallo, Hammer and Simeone proposed to further improve a feasible solution, through a sequence of *fill-up and exchange operations* as proposed by Peterson [373].

Hammer and Rader [210] presented a different primal heuristic, named *LEX*, based on the best *linear L_2-approximation* of (QKP) as presented in Hammer and Holzman [206]. The best linear approximation is

$$\begin{aligned} &\text{maximize} \sum_{j\in N} \pi^1_j x_j \\ &\text{subject to} \sum_{j\in N} w_j x_j \le c, \\ &\qquad x_j \in \{0,1\}, \quad j \in N, \end{aligned} \tag{12.93}$$

where π^1 is the upper plane defined by (12.13). In each step of the LEX algorithm, the item with the highest *efficiency* π^1_j/w_j is selected and the corresponding solution variable x_j is assigned the value 1. All items, which no longer fit into the residual capacity of the knapsack, are removed from the problem and their solution variables x_j are set to 0. The process is repeated until no items fit into the knapsack. In the last phase of the algorithm, local improvements are performed, by either exchanging items, or filling up with new items that fit into the current residual capacity. When choosing the items to exchange or include, first and second order "derivatives" are used, as in Gallo, Hammer and Simeone above.

A well-performing dual heuristic was presented by Billionnet and Calmels [37]. This algorithm first generates a greedy-type solution by initially setting $x_j = 1$ for $j \in N$, and then iteratively changing the value of a variable from 1 to 0, so as to achieve the smallest loss in the objective value, until a feasible solution is obtained. In the second step a sequence of iterations is performed in order to improve the solution by local exchanges. Let $S = \{j \in N | x_j = 1\}$ be the set of the items selected in the current solution. For each $j \in N \setminus S$, if $w_j + \sum_{\ell\in S} w_\ell \le c$ set $I_j = \emptyset$ and let the quantity δ_j be the objective function increase when x_j is set to 1. Otherwise, let δ_j be the largest profit increase when setting $x_j = 1$ and $x_i = 0$ for some $i \in S$ such that $w_j - w_i + \sum_{\ell\in S} w_\ell \le c$, and let $I_j = \{i\}$. Choosing k such that $\delta_k = \max_{j\in N\setminus S} \delta_j$, the heuristic algorithm terminates if $\delta_k \le 0$, otherwise the current solution is set to $S \setminus I_k \cup \{k\}$ and another iteration is performed.

Caprara, Pisinger and Toth [69] improved this bound by building it into the subgradient algorithm used for deriving bound $\hat{U}^2_{\text{CPT}}$ as described in Section 12.2.5. The

heuristic by Billionnet and Calmels is used at the first step of the algorithm, while at each iteration of the subgradient optimization procedure a heuristic solution is derived as follows. The LP-solution of (12.45) subject to (12.40)–(12.42) is rounded down, yielding an integer solution x. Starting from x the improvement part of the above algorithm is performed. The solutions obtained this way are typically substantially different from each other, even for slightly different Lagrangian profits, showing that the heuristic algorithm is worth to be applied often during the subgradient procedure.

It should also be mentioned that Glover and Kochenberger [183] presented a *tabu search* method for solving (QKP), based on a reformulation scheme.

12.7 Approximation Algorithms

Since (QKP) is strongly $\mathcal{NP}$-hard we cannot expect to find a fully polynomial approximation scheme unless $\mathcal{P} = \mathcal{NP}$. However, Rader and Woeginger [398] developed an *FPTAS* for the special case where all profits $p_{ij} \geq 0$ and where the underlying graph $G = (V,E)$ is so-called *edge series parallel*. The generalization of (QKP) to a graph $G = (V,E)$ is given by

$$\max \left\{ \sum_{j \in V} p_{jj} + \sum_{(i,j) \in E} p_{ij} x_i x_j \;\middle|\; \sum_{j \in V} w_j x_j \leq c, x_j \in \{0,1\}, j \in V \right\}.$$

The result relies on the fact that it is quite easy to develop a dynamic programming algorithm with this underlying structure.

Rader and Woeginger also prove that if the underlying graph $G = (V,E)$ is so-called *vertex series parallel*, then the problem is strongly $\mathcal{NP}$-hard, and hence we cannot expect to find an *FPTAS*. The proof is based on reduction from the *balanced complete bipartite subgraph problem* problem. The edge and vertex series parallel graphs are defined as follows:

Definition 12.7.1 *The class of* edge series parallel graphs *(ESP) is defined by the following three rules:*

a) Every graph consisting of two terminals connected by a single edge is ESP.

b) If two graphs G_1 and G_2 are ESP, then $(G_1 * G_2)$ is ESP, and $(G_1 || G_2)$ is ESP.

c) No other graphs than those defined by (a) and (b) are ESP.

Assuming that each graph G has two special vertices denoted the left and right terminal, the series composition *$(G_1 * G_2)$ results from identifying the right terminal of G_1 with the left terminal of G_2, having the left terminal of G_1 as new left terminal,*

and having the right terminal of G_2 as new right terminal. The parallel composition *$(G_1||G_2)$ results from identifying both right terminals respectively both left terminals with each other, and having these terminals as the new left and right terminal.*

An undirected graph is vertex series parallel *(VSP) if it is the underlying undirected graph of a digraph whose transitive closure equals the transitive closure of some minimal vertex series parallel graphs (MVSP), defined by the following three rules:*

a) Every graph consisting of a single vertex is MVSP.

b) If two digraphs G_1 and G_2 are MVSP, then $(G_1 * G_2)$ is MVSP, and $(G_1||G_2)$ is MVSP.

c) No other graphs than those defined by (a) and (b) are MVSP.

The series composition *$(G_1 * G_2)$ has vertex set $V_1 \cup V_2$ and arc set $E_1 \cup E_2 \cup (T_1 \times S_2)$ where T_1 is the set of sinks in G_1 and S_2 is the set of sources in G_2. The* parallel composition *$(G_1||G_2)$ has vertex set $V_1 \cup V_2$ and arc set $E_1 \cup E_2$. In a directed graph (*digraph*), a vertex whose in-degree is 0 is called a* source of a graph *and a vertex whose out-degree is 0 is called a* terminal of a graph.

Under the assumption that the profits may take on negative values, Rader and Woeginger [398] proved:

Proposition 12.7.2 *The (QKP) with positive and negative profits p_{ij} does not have any polynomial time approximation algorithm with fixed approximation ratio unless $\mathcal{P} = \mathcal{NP}$.*

Proof. The stated is proved by reduction from SSP-DECISION (see Appendix A.2 for a definition). Assume that some nonnegative integer weights $w'_1, \ldots, w'_n$ are given and a capacity c'. We construct an instance of (QKP) with $n+1$ items by choosing $w_0 := 0$, $w_j := w'_j$, $j = 1, \ldots, n$ and $c := c'$. The profits are $p_{00} := -c' + 1$ and $p_{0j} := w_j$ with all other profits set to 0.

A feasible solution to this (QKP) may either choose $x_0 = 0$ in which case the optimal solution value is $z^* = 0$. Otherwise, if $x_0 = 1$ then the solution value cannot exceed $z^* = -c + 1 + \sum_{j \in N} w_j x_j = 1$ due to the capacity constraint $\sum_{j \in N} w_j x_j \leq c$. The solution value $z^* = 1$ is attained if and only if SSP-DECISION has a feasible solution.

Now, assuming that an approximation algorithm with a bounded approximation ratio did exist for (QKP), we could use the algorithm to decide SSP-DECISION by solving the corresponding (QKP) and observing whether the approximate solution z is strictly positive. □

For the considered case (12.1) where profits are nonnegative—to the best knowledge of the authors—it is unknown whether (QKP) can be approximated with a constant approximation factor, see e.g. Crescenzi and Kann [94].

12.8 Computational Experiments — Exact Algorithms

Gallo, Hammer and Simeone [160] presented a family of randomly generated instances, which form also the benchmark for the algorithms by e.g. Billionnet and Calmels [37], Michelon and Veilleux [347] and Caprara, Pisinger and Toth [69].

The randomly generated instances by Gallo, Hammer and Simeone are constructed as follows. Let Δ be the *density* of the instance, i.e., the expected percentage of non-zero elements in the profit matrix P. Each weight w_j is randomly distributed in $[1,50]$ while the profits $p_{ij} = p_{ji}$ are nonzero with probability Δ, and in this case randomly distributed in $[1,100]$. Finally, the capacity c is randomly distributed in $[50, \sum_{j=1}^{n} w_j]$. Notice that Gallo, Hammer and Simeone actually chose the capacities in $[1, \sum_{j=1}^{n} w_j]$ but later papers have increased the lower limit.

In Table 12.1 we consider instances with up to 400 items solved by the QuadKnap algorithm. For each size n, the entries are average values of 10 instances. For each density, we report the results up to the highest value of n (multiple of 20) such that all 10 instances were solved to optimality within a time limit of 50000 seconds for each instance. The first entry gives the time, in seconds, used at the root node for deriving upper and lower bounds as well as reducing variables, while the next two columns give the percentage gap between the upper/lower bound and the optimal solution value at the root node. They are followed by the number of reduced variables. Finally, we give the number of branch-and-bound nodes investigated, and the average solution time in seconds.

One can observe that the upper and lower bounds are generally very tight, making it possible to reduce a majority of the variables, on average more than 75%. The total preprocessing takes a couple of minutes for the largest instances. Despite this effective preprocessing, the final branch-and-bound phase demands some hours and a huge number of nodes for the largest instances, as many variables have to be fixed by branching before closing the gap, despite the latter is typically very small already at the root node. Apparently the algorithm works best for high-density instances since the upper bounds are generally tighter in these cases. The lower bounds are in all cases nearly optimal. Quite different behaviors were observed on instances associated with the same Δ and n.

The solution times constitute a significant improvement with respect to previously published algorithms, in particular for instances with high density. The algorithm by Billionnet and Calmels [37] is only able to solve all the instances up to $n = 30$. The algorithm by Michelon and Veilleux [347] is slightly better, being able to solve all instances up to $n = 40$. According to Hammer and Rader [210], the largest instances solvable by the algorithm of Gallo, Hammer and Simeone [160] have size $n = 75$. Finally, the largest instances with density $\Delta = 100\%$ solved in Chaillou, Hansen and Mahieu [75] and Hammer and Rader [210] have size $n = 100$.

Δ	n	root time	gap.upper bound (%)	gap.lower bound (%)	reduced variables	b&b nodes	total time
	20	0.03	3.19	0.00	14	23	0.03
	40	0.20	2.64	0.00	20	350	0.20
25	60	0.36	0.45	0.00	46	119	0.36
	80	0.69	1.02	0.03	46	11367	0.84
	100	1.70	2.54	0.03	21	1605951	41.24
	120	3.25	0.44	0.00	70	9191	3.48
	20	0.03	4.09	0.00	14	23	0.03
	40	0.14	3.08	0.00	20	787	0.15
	60	0.30	1.98	0.06	27	1648	0.32
50	80	0.70	0.64	0.00	49	1391	0.72
	100	1.41	2.22	0.01	36	149888	4.80
	120	2.10	1.17	0.02	44	28172	2.63
	140	4.67	1.23	0.02	70	1354323	52.50
	160	6.83	0.70	0.00	82	186119	14.78
	20	0.03	4.21	0.00	13	24	0.03
	40	0.16	2.08	0.00	25	127	0.16
	60	0.37	1.04	0.00	44	194	0.37
	80	0.69	0.80	0.08	47	315	0.69
	100	1.20	1.94	0.00	52	1312	1.21
	120	1.46	0.65	0.00	85	1548	1.46
	140	3.20	0.82	0.01	73	13097	3.46
	160	4.66	0.73	0.05	73	22770	4.90
75	180	9.78	0.50	0.00	105	1612	9.83
	200	6.80	0.49	0.00	130	2769	6.86
	220	15.54	0.74	0.02	93	312734	20.97
	240	16.89	0.32	0.03	146	72534	18.79
	260	24.39	1.08	0.05	70	192434	38.26
	280	19.18	0.20	0.00	217	97351	19.48
	300	39.66	0.34	0.05	166	328454	46.12
	320	37.82	0.47	0.01	166	130712925	286.79
	340	59.15	0.25	0.03	204	294083	64.84
	360	68.97	0.92	0.02	144	8952166	1156.62
	20	0.03	4.91	0.00	12	57	0.03
	40	0.12	1.68	0.00	24	303	0.12
	60	0.27	1.04	0.00	43	425	0.27
	80	0.53	0.57	0.00	64	2167	0.53
	100	1.08	0.30	0.00	78	656	1.08
	120	1.95	0.56	0.00	79	31133	2.00
	140	4.16	0.32	0.00	103	44208	4.26
	160	3.33	0.33	0.00	128	46425	3.45
	180	6.73	0.27	0.17	133	360290	7.14
100	200	8.59	0.54	0.00	115	7882641	43.38
	220	9.33	0.19	0.05	158	17871673	144.52
	240	11.53	0.20	0.01	189	49128	11.85
	260	14.12	0.19	0.00	193	497389133	778.72
	280	27.38	0.17	0.00	207	21608076	347.59
	300	23.38	0.12	0.00	255	3605738	48.14
	320	39.80	0.23	0.00	211	240073849	376.22
	340	33.93	0.17	0.00	256	87743885	1411.21
	360	42.41	0.15	0.00	298	79584918	292.44
	380	54.56	0.17	0.00	278	500075295	2525.29
	400	61.15	0.10	0.00	323	413737707	1321.01

Table 12.1. Performances of the QuadKnap algorithm on randomly-generated instances with different densities Δ (INTEL PENTIUM 4, 1.5 GHz).

12.9 Computational Experiments — Upper Bounds

In this section we will experimentally compare the quality of the upper bounds presented in Section 12.2 with respect to tightness and computational effort. The experiments were run on an INTEL PENTIUM III, 933 MHz processor. The implementation and experimental study was carried out by Rasmussen and Sandvik [403]. We consider the classical (QKP) instances by Gallo, Hammer and Simeone [160] described in the previous section.

The performance of the bounds with respect to tightness and computational effort is reported in the following tables. The tightness of the bounds is measured in comparison to the optimal solution values z^*, which were obtained by the algorithm of Caprara, Pisinger and Toth [69]. In three instances it was not possible to find the optimal solution value z^* in reasonable time, in which case a lower bound found by the Lagrangian heuristic of [69] described in Section 12.6 is used. The difficult instances are found for $(n = 140, \Delta = 25)$, $(n = 200, \Delta = 25)$ and $(n = 180, \Delta = 50)$ where a single instance out of ten could not be solved.

The bound U^2_{BC} by Billionnet and Calmels was calculated using *CPLEX* 7.0. It was also used for solving the maximum flow problem in the bound U_{CHM} by Chaillou, Hansen and Mahieu and the bound $\hat{U}^2_{\text{MV}}$ by Michelon and Veuilleux. The bounds $U^0_{\text{HRW}}, U^1_{\text{HRW}}, U^2_{\text{HRW}}, U^3_{\text{HRW}}, U^4_{\text{HRW}}$ based on semidefinite programming were calculated using the *SeDuMi* 1.05 package by Sturm [447] for *Matlab*. The bounds $U^1_{\text{GHS}}, U^2_{\text{GHS}}, U^3_{\text{GHS}}, U^4_{\text{GHS}}$ by Gallo, Hammer and Simeone, as well as the bound $\hat{U}^2_{\text{CPT}}$ by Caprara, Pisinger and Toth and the bound $\hat{U}^2_{\text{MV}}$ by Michelon and Veuilleux, involve the solution of a knapsack problem. This problem is solved using the Minknap algorithm by Pisinger [383] described in Section 5.4.2, which also includes a routine for solving the continuous relaxation in $O(n)$. Lagrangian multipliers are calculated using Held and Karp [217] subgradient optimization.

When comparing solution times in the following tables, one should keep in mind that several of the bounds could have been implemented more efficiently, e.g. by using specialized algorithms for solving the maximum flow problems involved, or by using more efficient general solvers. However, the times still give an indication of the order of magnitude of the calculation times. What the tables do not show, is whether the bounds may be derived quicker inside a branch-and-bound algorithm, by reusing calculations from the previous bound. This information may be more important than the absolute calculation time, but only few of the bounds are described in the literature with focus on this aspect.

In Tables 12.2 to 12.4 we report the average values of 10 instances solved for each value of the density Δ and instance size n. For each bound we report the average solution time (`time`) in seconds, and the average deviation in percent from the optimal solution (`dev`). It can be seen that the bounds based on upper planes $U^1_{\text{GHS}}, U^2_{\text{GHS}}, U^3_{\text{GHS}}, U^4_{\text{GHS}}, \overline{U}^4_{\text{GHS}}$ are very fast to derive, but the quality of the bounds is disappointing. The bounds based on Lagrangian relaxation $U_{\text{CHM}}, \hat{U}^2_{\text{MV}}, \hat{U}^2_{\text{BFS}}$ are in gen-

		U^1_{GHS}		U^2_{GHS}		U^3_{GHS}		U^4_{GHS}		$\overline{U}^4_{\text{GHS}}$	
Δ	n	time	dev	time	dev	time	dev	time	dev	time	dev
5	40	0.000	6.38	0.002	6.38	0.001	6.38	0.000	6.38	0.000	6.38
	60	0.000	17.61	0.000	17.61	0.001	17.61	0.002	17.61	0.001	17.49
	80	0.000	22.70	0.000	22.70	0.000	22.70	0.001	22.70	0.001	22.70
	100	0.000	13.84	0.002	13.84	0.003	13.84	0.001	13.84	0.000	13.84
	120	0.001	25.04	0.001	25.04	0.002	25.03	0.001	25.02	0.001	25.02
	140	0.000	40.17	0.000	40.17	0.006	39.22	0.002	39.09	0.003	38.99
	160	0.001	21.53	0.001	21.53	0.003	21.32	0.005	21.27	0.005	21.24
	180	0.002	21.69	0.002	21.69	0.008	21.69	0.002	21.69	0.002	21.69
	200	0.002	18.06	0.004	18.06	0.008	18.06	0.002	18.06	0.004	18.06
	avrg	0.001	20.78	0.001	20.78	0.004	20.65	0.002	20.63	0.002	20.60
25	40	0.000	35.22	0.001	34.00	0.002	27.82	0.001	27.33	0.000	26.45
	60	0.000	21.47	0.000	20.99	0.000	18.08	0.001	17.88	0.001	17.76
	80	0.000	16.83	0.001	16.78	0.001	14.89	0.003	14.81	0.001	14.74
	100	0.000	61.73	0.003	61.36	0.002	57.57	0.000	57.41	0.001	57.17
	120	0.000	31.08	0.001	30.64	0.002	27.30	0.001	27.24	0.004	27.18
	140	0.002	36.89	0.002	36.89	0.004	35.66	0.001	35.62	0.005	35.57
	160	0.000	18.54	0.001	18.54	0.006	17.96	0.002	17.94	0.006	17.91
	180	0.000	31.80	0.002	31.44	0.008	29.96	0.003	29.95	0.005	29.92
	200	0.003	49.02	0.003	49.02	0.009	42.59	0.010	42.50	0.006	42.31
	avrg	0.001	33.62	0.002	33.29	0.004	30.20	0.002	30.08	0.003	29.89
50	40	0.000	35.76	0.000	34.91	0.001	26.01	0.001	25.48	0.000	24.95
	60	0.000	36.11	0.000	34.17	0.000	27.61	0.001	27.35	0.002	26.98
	80	0.000	18.48	0.000	18.39	0.003	14.86	0.002	14.76	0.001	14.64
	100	0.001	52.13	0.001	46.00	0.003	34.87	0.001	34.70	0.003	34.42
	120	0.001	31.47	0.005	29.80	0.003	22.59	0.002	22.51	0.003	22.36
	140	0.000	43.17	0.002	43.14	0.003	38.05	0.006	37.97	0.003	37.84
	160	0.000	23.59	0.002	22.41	0.003	16.80	0.004	16.76	0.006	16.67
	180	0.003	30.88	0.007	28.91	0.005	21.90	0.005	21.87	0.007	21.79
	200	0.000	29.32	0.005	28.60	0.010	23.23	0.007	23.20	0.007	23.15
	avrg	0.001	33.43	0.002	31.81	0.003	25.10	0.003	24.96	0.004	24.75
75	40	0.000	35.10	0.000	28.80	0.000	18.48	0.000	18.00	0.000	17.55
	60	0.001	37.38	0.000	29.12	0.000	17.36	0.000	17.06	0.002	16.73
	80	0.000	26.77	0.001	23.04	0.002	15.37	0.003	15.27	0.000	15.12
	100	0.000	31.20	0.001	22.89	0.004	13.85	0.002	13.77	0.002	13.63
	120	0.000	33.20	0.002	21.57	0.002	11.59	0.003	11.53	0.003	11.39
	140	0.000	36.53	0.002	27.27	0.005	16.80	0.005	16.75	0.004	16.66
	160	0.000	25.79	0.002	22.07	0.004	16.13	0.007	16.11	0.007	16.05
	180	0.002	52.66	0.006	34.91	0.006	20.86	0.009	20.82	0.006	20.71
	200	0.001	51.06	0.004	26.00	0.007	13.30	0.009	13.26	0.009	13.19
	avrg	0.000	36.63	0.002	26.19	0.003	15.97	0.004	15.84	0.004	15.67
95	40	0.000	63.84	0.001	35.23	0.003	17.33	0.000	16.72	0.000	16.15
	60	0.000	61.20	0.000	35.03	0.002	16.16	0.001	15.86	0.003	15.51
	80	0.000	39.89	0.001	24.19	0.000	12.88	0.002	12.74	0.002	12.59
	100	0.002	49.27	0.002	27.00	0.001	13.50	0.006	13.39	0.000	13.28
	120	0.002	27.16	0.002	18.52	0.002	10.64	0.004	10.59	0.004	10.52
	140	0.002	28.02	0.004	19.79	0.006	11.69	0.003	11.65	0.005	11.58
	160	0.000	34.82	0.002	23.34	0.004	12.11	0.008	12.08	0.008	12.01
	180	0.002	31.71	0.004	20.00	0.006	11.27	0.008	11.24	0.003	11.18
	200	0.002	53.23	0.003	31.67	0.009	14.33	0.007	14.30	0.015	14.21
	avrg	0.001	43.24	0.002	26.08	0.004	13.32	0.004	13.17	0.004	13.00
100	40	0.001	23.67	0.001	15.24	0.001	9.64	0.001	9.32	0.000	9.15
	60	0.000	19.93	0.000	11.09	0.001	6.09	0.001	5.96	0.001	5.88
	80	0.000	40.13	0.000	21.15	0.001	10.72	0.003	10.58	0.001	10.45
	100	0.001	33.11	0.001	19.42	0.003	10.46	0.003	10.38	0.002	10.29
	120	0.001	37.67	0.004	22.86	0.002	11.33	0.003	11.27	0.005	11.19
	140	0.000	26.21	0.002	17.50	0.006	10.35	0.004	10.31	0.004	10.25
	160	0.000	38.14	0.001	19.63	0.007	9.68	0.006	9.64	0.005	9.59
	180	0.002	46.30	0.005	24.20	0.005	9.86	0.007	9.83	0.009	9.77
	200	0.003	40.76	0.004	22.01	0.009	10.09	0.010	10.07	0.008	10.01
	avrg	0.001	33.99	0.002	19.23	0.004	9.80	0.004	9.71	0.004	9.62
total avrg		0.001	33.62	0.002	26.23	0.004	19.18	0.003	19.06	0.003	18.92

Table 12.2. Bounds from upper planes (INTEL PENTIUM III, 933 MHz).

Δ	n	U_{CHM} time	U_{CHM} dev	$\hat{U}^2_{MV}$ time	$\hat{U}^2_{MV}$ dev	$\hat{U}^2_{BFS}$ time	$\hat{U}^2_{BFS}$ dev
5	40	0.0	0.80	0.3	0.27	3.6	0.74
	60	0.0	0.65	0.6	0.37	9.6	0.49
	80	0.0	0.45	1.4	0.29	15.8	0.43
	100	0.0	0.32	2.0	0.16	24.8	0.31
	120	0.0	0.28	3.2	0.17	40.4	0.30
	140	0.0	0.56	2.2	0.53	75.6	0.70
	160	0.0	0.13	12.5	0.08	76.5	0.14
	180	0.0	0.14	7.3	0.11	95.0	0.17
	200	0.1	0.08	22.0	0.05	111.0	0.09
	avrg	0.0	0.38	5.7	0.22	50.2	0.37
25	40	0.0	3.07	0.6	2.38	3.7	1.65
	60	0.0	0.58	1.8	0.37	8.5	0.35
	80	0.0	1.02	6.5	0.77	16.1	0.65
	100	0.0	2.48	3.0	2.44	30.0	0.68
	120	0.1	0.57	4.9	0.53	38.6	0.26
	140	0.1	1.46	2.4	1.46	59.4	0.61
	160	0.2	0.50	64.7	0.47	62.2	0.16
	180	0.2	0.74	34.8	0.73	93.5	0.27
	200	0.3	1.47	20.3	1.47	122.3	0.36
	avrg	0.1	1.32	15.5	1.18	48.3	0.55
50	40	0.0	3.70	1.1	3.33	3.7	1.33
	60	0.0	2.85	3.1	2.70	8.4	0.67
	80	0.0	0.82	9.4	0.69	14.7	0.38
	100	0.1	3.61	15.4	3.57	24.7	0.50
	120	0.2	1.40	48.5	1.35	36.1	0.35
	140	0.3	1.33	28.4	1.32	47.3	0.19
	160	0.5	1.16	164.9	1.12	66.1	0.20
	180	0.6	1.57	27.9	1.57	84.8	0.21
	200	0.8	0.83	43.8	0.83	107.0	0.16
	avrg	0.3	1.92	38.1	1.83	43.7	0.44
75	40	0.0	3.92	3.3	3.36	3.7	1.58
	60	0.0	2.15	6.4	1.88	8.1	0.81
	80	0.1	1.52	2.0	1.52	13.5	0.23
	100	0.2	1.96	15.1	1.93	23.3	0.41
	120	0.3	1.89	195.5	1.80	31.2	0.49
	140	0.5	1.94	30.6	1.94	47.5	0.19
	160	0.7	1.24	506.8	1.21	63.1	0.18
	180	0.8	1.80	14.4	1.80	77.8	0.12
	200	3.7	2.03	17.9	2.03	102.5	0.17
	avrg	0.7	2.05	88.0	1.94	41.2	0.46
95	40	0.0	9.30	0.4	9.28	3.8	1.60
	60	0.0	4.21	0.6	4.21	8.4	0.68
	80	0.1	2.59	11.2	2.54	14.6	0.71
	100	0.2	1.63	13.1	1.62	21.0	0.46
	120	0.4	1.71	89.4	1.69	32.7	0.28
	140	0.5	1.82	113.2	1.81	41.5	0.22
	160	0.8	2.18	221.6	2.18	62.9	0.33
	180	1.0	2.06	15.1	2.06	80.8	0.32
	200	1.2	3.38	563.3	3.37	103.7	0.15
	avrg	0.5	3.21	114.2	3.20	41.0	0.53
100	40	0.0	4.22	2.0	4.08	3.6	1.55
	60	0.0	2.14	11.2	2.05	8.2	0.99
	80	0.1	2.22	20.4	2.20	14.0	0.53
	100	0.2	2.90	4.4	2.90	22.0	0.27
	120	0.4	2.23	20.0	2.22	33.1	0.38
	140	0.6	1.93	122.7	1.92	47.9	0.26
	160	0.8	2.25	978.7	2.21	60.0	0.31
	180	1.0	2.54	21.1	2.54	78.3	0.24
	200	1.4	1.66	79.8	1.66	103.1	0.50
	avrg	0.5	2.45	140.0	2.42	41.1	0.56
total avrg		0.3	1.89	66.9	1.80	44.3	0.49

Δ	n	$\hat{U}^2_{CPT}$ time	$\hat{U}^2_{CPT}$ dev	U^2_{BC} time	U^2_{BC} dev
5	40	0.0	1.23	25.2	0.70
	60	0.1	1.41	160.0	0.45
	80	0.2	1.56		
	100	0.4	1.32		
	120	0.7	1.37		
	140	1.5	2.15		
	160	2.1	1.49		
	180	3.7	1.64		
	200	4.8	1.44		
	avrg	1.5	1.51	92.6	0.57
25	40	0.0	2.91	35.5	2.21
	60	0.1	0.92	326.4	0.38
	80	0.3	1.34		
	100	0.7	2.79		
	120	1.0	0.93		
	140	2.1	1.96		
	160	2.5	1.01		
	180	4.5	1.18		
	200	7.7	1.91		
	avrg	2.1	1.66	181.0	1.30
50	40	0.1	3.29	55.9	1.70
	60	0.2	2.11	699.3	0.84
	80	0.3	0.85		
	100	0.7	2.40		
	120	1.2	1.31		
	140	2.2	1.45		
	160	2.9	0.89		
	180	5.2	1.57		
	200	6.0	0.84		
	avrg	2.1	1.64	377.6	1.27
75	40	0.1	2.45	29.4	1.59
	60	0.1	1.39	116.4	0.80
	80	0.3	1.03	978.8	0.22
	100	0.5	1.21	2110.9	0.41
	120	0.8	0.73	2982.4	0.47
	140	1.7	1.11		
	160	2.5	0.84		
	180	3.6	0.78		
	200	4.2	0.78		
	avrg	1.5	1.15	1243.6	0.70
95	40	0.1	2.69	17.7	1.59
	60	0.1	1.25	102.2	0.68
	80	0.3	1.12	445.8	0.70
	100	0.4	0.91	1346.5	0.46
	120	0.7	0.66		
	140	1.3	0.65		
	160	2.1	0.63		
	180	3.2	0.71		
	200	4.7	0.54		
	avrg	1.4	1.02	478.0	0.86
100	40	0.1	1.89	24.8	1.55
	60	0.1	1.17	151.1	0.98
	80	0.3	0.74	397.7	0.51
	100	0.5	0.61	1990.1	0.27
	120	0.9	0.74		
	140	1.3	0.53		
	160	1.9	0.50		
	180	2.9	0.46		
	200	4.2	0.66		
	avrg	1.3	0.81	640.9	0.83
total avrg		1.7	1.30	631.4	0.87

Table 12.3. Bounds based on Lagrangian relaxation (left table) and bounds from Linearisation (right table). (INTEL PENTIUM III, 933 MHz).

		U^0_{HRW}		U^1_{HRW}		U^2_{HRW}		U^3_{HRW}		U^4_{HRW}	
Δ	n	time	dev	time	dev	time	dev	time	dev	time	dev
5	40	3.9	3.40	5.6	1.29	5.3	1.22	5.9	1.22	14.2	1.21
	60	9.7	7.64	12.4	2.70	13.5	1.84	14.2	1.83	75.8	1.83
	80	25.5	10.23	29.1	3.60	34.2	1.95	36.8	1.95	222.1	1.95
	100	54.3	6.82	68.1	2.24	83.5	1.12	90.6	1.12	579.5	1.12
	120	125.4	11.02	143.0	4.19	170.4	1.58	180.1	1.58	1053.8	1.58
	140	243.1	17.97	273.7	8.24	309.4	2.47	327.9	2.47	1810.4	2.47
	160	408.0	10.42	509.7	3.95	566.3	1.03	619.8	1.03	2410.6	1.03
	180	644.5	11.45	799.2	5.01	909.3	1.19	1014.2	1.19	3928.0	1.19
	200	904.3	9.76	1107.9	3.43	1345.5	0.69	1564.6	0.69		
	avrg	268.8	9.86	327.6	3.85	381.9	1.46	428.2	1.45	1261.8	1.55
25	40	3.8	17.36	4.2	10.44	4.7	2.99	5.4	2.97	17.8	2.95
	60	10.0	11.79	11.5	5.07	14.8	0.71	15.3	0.70	69.6	0.70
	80	24.4	9.78	28.7	4.96	38.2	0.87	42.1	0.86	255.5	0.85
	100	55.2	31.42	61.1	18.31	72.0	1.23	80.6	1.22	537.0	1.21
	120	120.3	16.75	138.6	8.77	185.1	0.49	204.2	0.48	1090.5	0.47
	140	240.7	21.13	274.3	11.59	340.3	0.49	350.7	0.49	1968.3	0.49
	160	373.4	11.62	457.7	5.78	626.3	0.25	679.3	0.25	3146.9	0.25
	180	583.3	18.92	664.0	10.00	888.5	0.31	929.8	0.30	4885.8	0.30
	200	916.2	24.76	1019.1	14.33	1306.5	0.34	1320.2	0.34		
	avrg	258.6	18.17	295.5	9.92	386.3	0.85	403.1	0.85	1496.4	0.90
50	40	3.8	20.46	4.1	12.23	5.1	1.85	5.7	1.83	17.3	1.82
	60	10.1	21.03	11.0	12.16	13.9	1.03	15.7	1.01	76.0	0.99
	80	23.0	11.22	27.5	5.86	39.6	0.48	39.4	0.48	247.7	0.48
	100	55.1	27.69	61.0	16.58	79.2	0.66	87.7	0.63	576.6	0.61
	120	121.9	17.60	136.7	10.02	193.2	0.40	206.4	0.39	1185.8	0.38
	140	240.2	25.51	265.6	14.00	396.7	0.23	397.2	0.23	2370.4	0.23
	160	397.0	13.32	449.8	7.54	679.1	0.23	750.9	0.21	3333.8	0.21
	180	627.1	17.62	704.2	10.11	1022.0	0.21	1113.6	0.20	4668.1	0.20
	200	855.6	17.21	947.6	9.39	1399.9	0.13	1576.1	0.13		
	avrg	259.3	19.07	289.7	10.88	425.4	0.58	465.9	0.57	1559.5	0.61
75	40	4.0	21.46	4.0	12.94	5.1	1.88	5.4	1.87	19.8	1.87
	60	9.5	20.72	10.8	11.82	14.3	1.01	16.4	0.98	72.8	0.97
	80	22.5	16.06	25.6	9.05	41.1	0.28	42.3	0.27	241.4	0.27
	100	53.9	16.96	61.0	10.15	89.8	0.47	94.0	0.47	628.0	0.46
	120	114.6	16.19	126.7	9.88	185.7	0.68	210.3	0.65	1254.4	0.64
	140	236.0	19.73	262.4	11.88	406.1	0.24	447.8	0.22	2168.5	0.21
	160	382.5	15.76	453.0	8.68	713.6	0.21	749.2	0.20	3191.2	0.20
	180	651.2	25.19	697.3	15.25	1064.5	0.35	1358.0	0.31		
	200	913.6	20.60	963.6	13.32	1382.2	0.44	2036.5	0.36		
	avrg	265.3	19.19	289.4	11.44	433.6	0.61	551.1	0.59	1082.3	0.66
95	40	3.7	38.28	3.9	25.05	4.9	1.80	5.3	1.78	17.3	1.78
	60	9.4	34.16	10.7	20.90	14.9	0.82	16.6	0.79	75.6	0.78
	80	23.2	22.26	26.2	13.48	40.8	0.78	43.8	0.76	256.5	0.75
	100	54.6	24.82	59.6	14.78	96.8	0.65	102.5	0.61	655.5	0.59
	120	117.1	16.32	131.7	9.32	228.7	0.34	233.0	0.32	1248.6	0.31
	140	236.5	17.29	259.9	9.82	428.0	0.27	464.6	0.27	2165.8	0.27
	160	369.1	21.00	419.3	12.44	729.5	0.34	732.6	0.34	3426.4	0.34
	180	619.4	18.34	662.9	11.02	1182.8	0.37	1227.4	0.36	5121.6	0.35
	200	906.3	29.32	967.7	18.20	1678.9	0.16	1762.3	0.16		
	avrg	259.9	24.64	282.4	15.00	489.5	0.61	509.8	0.60	1620.9	0.65
100	40	3.7	17.04	4.1	10.59	5.8	1.66	5.9	1.66	21.1	1.66
	60	9.3	12.07	10.5	7.22	16.2	1.05	15.9	1.05	83.8	1.04
	80	22.3	22.30	25.8	13.17	38.1	0.59	41.5	0.58	258.0	0.57
	100	50.5	19.85	57.3	12.25	93.8	0.30	100.5	0.29	639.7	0.29
	120	119.5	22.26	130.4	13.04	238.6	0.40	235.2	0.40	1257.3	0.40
	140	238.9	16.86	260.2	9.61	459.0	0.27	457.8	0.26	2203.8	0.26
	160	405.8	20.41	442.0	12.41	717.2	0.39	803.8	0.38	3503.7	0.37
	180	620.9	24.25	678.2	14.85	1133.7	0.27	1210.9	0.26	4833.8	0.25
	200	893.2	21.91	970.0	13.16	1620.9	0.54	1757.8	0.53		
	avrg	262.7	19.66	286.5	11.81	480.4	0.61	514.4	0.60	1600.1	0.60
total avrg		262.4	18.43	295.2	10.48	432.8	0.79	478.7	0.78	1444.4	0.83

Table 12.4. Bounds from Semidefinite Programming. (INTEL PENTIUM III, 933 MHz).

eral much tighter, but the computational times are larger. The two bounds from linearisation $\hat{U}^2_{\text{CPT}}, U^2_{\text{BC}}$ are in general both quite tight, but U^2_{BC} is very time consuming, and hence it can only be calculated for small instances. Finally the bounds based on semidefinite programming show that the bounds $U^0_{\text{HRW}}, U^1_{\text{HRW}}$ are of poor quality, while $U^2_{\text{HRW}}, U^3_{\text{HRW}}, U^4_{\text{HRW}}$ more or less are of the same quality, but demanding increasing computational time.

The tightest of all bounds appears to be the bound $\hat{U}^2_{\text{BFS}}$ although it is also one of the most expensive to derive as it demands an enumerative search. This means in particular, that it is not well suited for use inside a branch-and-bound algorithm since it is hardly possible to reuse calculations from previous branching nodes. The bound U_{CHM} by Chaillou, Hansen and Mahieu is on the other hand cheap to derive and has a good quality, in particular for low densities. For high densities the bound $\hat{U}^2_{\text{CPT}}$ by Caprara, Pisinger and Toth has a good quality and it is also relatively cheap to derive.

For variable reduction as described in Section 12.3, one may use some of the more expensive bounds, since only $O(n)$ calculations of the bounds are needed. Possibly, one could start by using the cheaper bounds, and gradually apply tighter bounds on each remaining set of variables.

13. Other Knapsack Problems

In this chapter we consider knapsack type problems which have not been investigated in the preceding chapters. There is a huge amount of different kinds of variations of the knapsack problem in the scientific literature, often a specific problem is treated in only one or two papers. Thus, we could not include every knapsack variant but we tried to make a representative selection of interesting problems. Two problems will be presented in the first two sections more detailed. In Section 13.1 we start with multiobjective knapsack problems and and continue in Section 13.2 with results about precedence constraint knapsack problems. Finally, Section 13.3 contains a collection of several other variations of knapsack problems and the main results for these problems are mentioned.

13.1 Multiobjective Knapsack Problems

In this section we will consider knapsack problems with several objective functions. We start with a a short introduction into the basic notion of multicriteria optimization and a formal definition of multiobjective knapsack problem.

13.1.1 Introduction

Consider an optimization problem with t objective functions $V_1, \ldots, V_t$ and let I be an instance of this problem. If S is a feasible solution, then $V_k(S)$ denotes the value of S with respect to the k-th objective function, $1 \leq k \leq t$. Unless stated otherwise assume that all objectives are maximization criteria.

Computing a set of solutions which covers all possible trade-offs between the different objectives can be understood in different ways. We will define this (as most commonly done) as searching for "efficient" solutions. A feasible solution S_1 *weakly dominates* a feasible solution S_2 if

$$V_k(S_1) \geq V_k(S_2), \quad 1 \leq k \leq t. \tag{13.1}$$

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

A weakly dominating solution S_1 even *dominates* solution S_2 if at least one of the inequalities (13.1) is strict. A feasible solution S is *efficient* or *Pareto optimal* if there is no other feasible solution which dominates S. The set $\mathcal{P}$ of efficient solutions for I is called *Pareto frontier* (sometimes also *Pareto curve*). A set of feasible solutions is called *reduced* if it does not contain two different solutions S_1 and S_2 such that S_1 weakly dominates S_2. Usually, a Pareto frontier is assumed to be reduced. A survey and broader discussion of different concepts of multiobjective optimization can be found in the textbook by Ehrgott and Gandibleux [128].

The *multiobjective knapsack problem* (MOKP) is obtained from the classical knapsack problem by introducing t profit values instead of one for every item. More precisely, an instance I of (MOKP) consists of a set of items $N = \{1,\ldots,n\}$ and a knapsack with capacity c. Each item j ($j = 1,\ldots,n$) has t profits p_{kj} ($k = 1,\ldots,t$) and one weight w_j.

The *multiobjective d-dimensional knapsack problem* (MOd-KP) generalizes the d-dimensional knapsack problem (d-KP) introduced in Chapter 9. An instance consists of a set of items $N = \{1,\ldots,n\}$ and a knapsack with d capacities c_i. Each item j ($j = 1,\ldots,n$) has d weights w_{ij}, $i = 1,\ldots,d$, and t profits p_{kj}, $k = 1,\ldots,t$.

A feasible solution of (MOKP) is a subset $S \subseteq N$ satisfying the constraint $\sum_{j\in S} w_j \leq c$ corresponding to inequalities (1.2) and (1.3). Analogously, a feasible solution of (MOd-KP) is a subset $S \subseteq N$ satisfying the d constraints

$$\sum_{j\in S} w_{ij} \leq c_i \quad i = 1,\ldots,d,$$

corresponding to inequalities (9.2) and (9.3). In both cases, the t objective values of a feasible solution S are $V_1(S), \ldots, V_t(S)$ with $V_k(S) = \sum_{j\in S} p_{kj}$. Throughout this section, we will assume that both t and d are constants.

There is a wide range of applications for multiobjective knapsack problems. Bhaskar [35] and Rosenblatt and Sinuany-Stern [409] treat the *capital budgeting problem* with two objectives where the first objective is maximizing the present value of accepted projects and the second is minimizing their risk. Kwak et al. [291] investigate a capital budgeting problem with four criteria. Teng and Tzeng [454] present a transportation investment problem which can be modelled as (MOKP) with four objectives. For more details on financial decision problems we refer to Section 15.4. Applications on relocation issues in conservation biology have been described by Kostreva et al. [287]. Finally, Jenkins [249] investigates a bicriteria knapsack problem for planning remediation of contaminated lightstation sites.

The section on (MOKP) is organized as follows: In 13.1.2 we present exact algorithms for multiobjective knapsack problems. Section 13.1.3 lists general properties of approximation algorithm for multiobjective optimization problems. The following two subsections contain fully polynomial time approximation schemes and polynomial time approximation schemes. We finish in 13.1.6 with a collection of results for metaheuristics.

13.1.2 Exact Algorithms for (MOKP)

An exact algorithm with pseudopolynomial running time for (MOKP) can be easily obtained through dynamic programming by profits. This will be done by adapting algorithm DP-Profits from Section 2.3. To generalize this algorithm to the multiobjective case we can basically expand it to the t-dimensional profit space. First of all we have to compute upper bounds on each of the t objective functions. They can be obtained easily, e.g. by running Ext-Greedy separately for every objective. Thus, t upper bounds $U_1,\ldots,U_t$ are computed. For convenience let $U_{\max} := \max\{U_1,\ldots,U_t\}$.

Then we define the following dynamic programming function for $v_k = 0,1,\ldots,U_k$ and $k = 1,\ldots,t$:

$$y(v_1,\ldots,v_t) = w \qquad \Longleftrightarrow$$

there exists a subset of items with profit v_k in the k-th objective for $k = 1,\ldots,t$ and weight w, where w is minimal among all such sets.

After setting all function values to $c+1$ in the beginning (indicating that the corresponding combination of profits can not be reached yet) except $y(0,\ldots,0)$ which is set to 0, the function y can be computed by considering the items in turn and performing for every item j the update operation analogous to (2.10) for all feasible combinations of profit values (i.e. $v_k + p_{kj} \leq U_k$):

$$y(v_1+p_{1j},\ldots,v_t+p_{tj}) = \min\{y(v_1+p_{1j},\ldots,v_t+p_{tj}), y(v_1,\ldots,v_t)+w_j\} \quad (13.2)$$

As above this operation can be seen as trying to improve the current weight of an entry by adding item j to the entry $y(v_1,\ldots,v_t)$.

It remains to extract the Pareto frontier from y. This can be done in a straightforward way: Each entry of y satisfying $y(v_1,\ldots,v_t) \leq c$ corresponds to a feasible solution with objective values $v_1,\ldots,v_t$. The set of items leading to that solution can be determined easily if standard bookkeeping techniques are used. The feasible solutions are collected and the resulting set of solutions is reduced by discarding solutions that are weakly dominated by other solutions in the set.

The correctness of this approach follows from classical dynamic programming theory analogously to to the argumentation in Section 2.3. Its running time is given by going through the complete profit space for every item which is $O(n\prod_{k=1}^{t} U_k)$ and hence $O(n(U_{\max})^t)$.

In order to obtain lower computation times one can focus on the determination of an important subset of the efficient solutions, the so-called *supported efficient solutions*. A solution of a (MOKP) is called *supported* if it can be found through *weighted sum scalarization* (WSS), i.e. if it is a convex combination of the t objective functions, otherwise it is called *non-supported*. Formally the supported efficient solutions are the solutions of the following parametric integer linear program:

$$
\begin{aligned}
(WSS) \qquad \text{maximize} \quad & \sum_{k=1}^{t} \lambda_k \left(\sum_{j=1}^{n} p_{kj} x_j \right) \\
\text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\
& \sum_{k=1}^{t} \lambda_k = 1, \\
& x_j \in \{0,1\}, \quad j = 1,\ldots,n, \\
& \lambda_k \geq 0, \quad k = 1,\ldots,t.
\end{aligned}
$$

Rosenblatt and Sinuany-Stern [409] construct the efficient supported solutions for (MOKP) with two objectives by complete enumeration. Moreover, they introduce a greedy-like heuristic with better running time. Their approach was improved by Eben-Chaime [127]. He used the standard network representation for formulating the classical knapsack problem for a given instance I as a longest path problem:

The network $\mathcal{N}$ is built of $n+2$ layers. Layers 0 and $n+1$ consist of single nodes v_0 and v_{n+1}, respectively. Layers $1,\ldots,n$ correspond to the items $1,\ldots,n$ and have $c+1$ nodes each: More precisely, layer j has $c+1$ nodes $(0,j),\ldots,(c,j)$ for $j = 1,\ldots,n$.

There are only arcs between neighboring layers. We have arcs with profit zero from nodes (i,j) to $(i,j+1)$, $i = 0,\ldots,c$, $j = 1,\ldots,n-1$ and from nodes (i,n) to v_{n+1}, $i = 0,\ldots,c$. These arcs symbolize that item j is not included in the knapsack. Then we have arcs with profit p_j from nodes (i,j) to $(i+w_j,j+1)$ if $i+w_j \leq c$, $i = 1,\ldots,c$, $j = 1,\ldots,n-1$, and from nodes $(0,n),\ldots,(c-w_n,n)$ to v_{n+1}. These arcs symbolize that item j is included in the knapsack if and only if it fits into it. Finally, there are $c+1$ dummy arcs with profit zero from node v_0 to nodes $(0,1),\ldots,(c,1)$.

The size of network $\mathcal{N}$ can be reduced by constructing it recursively backwards starting with node v_{n+1} keeping only nodes $(c-w_n,n)$ and (c,n) in layer n, and so on. It can be easily seen that every path from v_0 to v_n in this network corresponds to a feasible solution for the given instance I. A longest path corresponds to an optimal solution. Due to the special structure of network $\mathcal{N}$ such a longest path can be found in time $O(nc)$. Applying methods for finding parametric solutions in networks, Eben-Chaime could collect all efficient supported solutions of a bicriteria knapsack problem in time $O(Knc)$. Parameter K represents the number of segments in a piecewise linear parametric solution.

However, Visée et al. [474] illustrated by numerical examples that the number of supported solutions of (MOKP) grows linearly with the number of items, while the number of non-supported solutions grows exponentially. Ulungu and Teghem [462] proposed a two-phase approach to construct *all* efficient solutions for a biobjective

knapsack problem. In the first phase the supported efficient solutions are found using weighted sum scalarization. In the second phase a branch-and-bound procedure is applied. An extension to t objectives is discussed. The second phase was improved in the paper by Visée et al. [474]. They presented two branch-and-bound approaches based on breadth first search and depth first search, respectively.

Captivo et al. [70] applied dynamic programming for solving general (MOKP). They started with the network $\mathcal{N}$ as described above. Then, their algorithm is based on a particular implementation of labeling algorithms for multiple shortest path problems.

Klamroth and Wiecek [272] gave a detailed study of different dynamic programming approaches for calculating the optimal solution of the *unbounded* multiobjective knapsack problem. They also used these dynamic programming recursions for solving a time-dependent multiobjective knapsack problem [273] and for a time-dependent single-machine scheduling problem [274].

13.1.3 Approximation of the Multiobjective Knapsack Problem

First we have to define what we mean by an approximation algorithm with relative performance guarantee for multiobjective optimization problems. Consider again an instance I of an optimization problem with t objective functions $V_1, \ldots, V_t$. A feasible solution S_1 is called a $(1-\varepsilon)$-*approximation algorithm* of a solution S_2 if $V_k(S_1) \geq (1-\varepsilon)V_k(S_2)$ for all $1 \leq k \leq t$. A set $\mathcal{F}$ of feasible solutions for I is called a $(1-\varepsilon)$-*approximation of the Pareto frontier* if, for every feasible solution S, the set $\mathcal{F}$ contains a feasible solution S' that is a $(1-\varepsilon)$-approximation of S.

The definition of *FPTAS* and *PTAS* is analogous to Section 2.6. An algorithm that always outputs a $(1-\varepsilon)$-approximation of the Pareto frontier is called a $(1-\varepsilon)$-*approximation algorithm*. A *PTAS for the Pareto frontier* is an algorithm which outputs for every input $\varepsilon > 0$, a $(1-\varepsilon)$-approximation algorithm A_ε and runs in polynomial time in the size of the input. If the running-time of A_ε is polynomial in the size of the input and in $\frac{1}{\varepsilon}$, the algorithm is called an *FPTAS*.

The Pareto frontier of an instance of a multiobjective optimization problem may contain an arbitrarily large number of solutions. On the contrary, for every $\varepsilon > 0$ there exists a $(1-\varepsilon)$-approximation of the Pareto frontier that consists of a number of solutions that is polynomial in the size of the instance and in $\frac{1}{\varepsilon}$ (under reasonable assumptions). An explicit proof for this observation was given by Papadimitriou and Yannakakis [368]. Consequently, a *PTAS* for a multiobjective optimization problem does not only have the advantage of computing a provably good approximation in polynomial time, but also has a good chance of presenting a reasonably small set of solutions to the user. Naturally, an *FPTAS* for a multiobjective optimization problem always outputs a solution which is polynomially bounded in the size of the instance and in $\frac{1}{\varepsilon}$.

An extensive study of multiobjective combinatorial optimization problems was carried out by Safer and Orlin [413, 414]. In [413], they study necessary and sufficient conditions for the existence of an *FPTAS* for a multiobjective optimization problem. Their approach is based on the concept of VPP algorithms and VPP reductions, where VPP stands for *value-pseudo-polynomial*. An algorithm for a multiobjective optimization problem is a VPP algorithm if it computes a Pareto frontier for a given instance I in time polynomial in the size of I and in $M_V(I)$, where $M_V(I)$ is essentially the largest value of any feasible solution with respect to any of the objectives. Safer and Orlin prove that an *FPTAS* for a problem exists if there is a VPP algorithm for the problem and if the objective functions are quasi-closed under scaling and quasi-closed under box constraints (which holds for many functions that are encountered in practice). Furthermore, a VPP algorithm for one problem can be obtained using a VPP reduction to another problem for which a VPP algorithm has already been found. In [414], they apply these techniques to obtain an *FPTAS* for various multiobjective network flow, knapsack, and scheduling problems. In particular, they prove the existence of an *FPTAS* for the multiobjective knapsack problem. However, it should be noted that their *FPTAS* is obtained using a VPP reduction, which in general does not lead to an *FPTAS* with a good running-time. In particular, their approach involves the solution of many different scaled versions of the given instance using a VPP algorithm.

Further results concerning the existence of an *FPTAS* are given by Papadimitriou and Yannakakis [368]. For a multiobjective optimization problem with t maximization objectives, they define the corresponding *gap problem* as follows: Given an instance of the multiobjective problem, a parameter $\varepsilon > 0$, and a vector $(v_1, \ldots, v_t)$, either return a solution S with $V_k(S) \geq v_k$ for $k = 1, \ldots, t$, or assert that no solution S' with $V_k(S') \geq (1 - \varepsilon)v_k$ for all k exists. Then they show that an *FPTAS* exists if and only if the corresponding gap problem can be solved efficiently. Their result can be stated as follows.

Theorem 13.1.1 *There is an* FPTAS *for an optimization problem with t objectives if and only if the corresponding gap problem can be solved in time polynomial in the size of the instance and in* $1/\varepsilon$.

This result was further exploited by Papadimitriou and Yannakakis to give a sufficient condition for the efficient approximability of multiobjective linear discrete optimization problems. Roughly speaking, these are problems where the feasible solutions are nonnegative integer vectors of a given length n and where the objective functions are of the form $\sum_{j=1}^{n} p_{kj}x_j$. The *exact version* of an discrete optimization problem is the following: Given an instance of the problem and an integer B, determine whether there is a feasible solution with cost exactly B.

Theorem 13.1.2 *There is an* FPTAS *for a linear discrete optimization problem with t objectives, if there is a pseudopolynomial algorithm for the exact version of the same problem.*

13.1.4 An *FPTAS* for the Multiobjective Knapsack Problem

Since the knapsack problem (KP) can be solved to optimality in pseudopolynomial time by dynamic programming, the existence of an *FPTAS* for (MOKP) follows immediately from Theorem 13.1.2. But the general construction used in the proof of Theorem 13.1.2 is again based on solving many different scaled versions of the problem and, therefore, less efficient than the specialized algorithm developed by Erlebach, Kellerer and Pferschy [132] which we present in this section.

To make the extension of an *FPTAS* from one to t objectives possible, we require an algorithm which computes a feasible solution with a relative ε-error for every possible profit value in every objective. In [132] an *FPTAS* for (KP) is presented which fulfills this particular property and it is shown how this *FPTAS* can be extended to an *FPTAS* for (MOKP).

The description of this *FPTAS* that is derived from the algorithm DP-Profits from Section 2.3 is relatively simple. The profit space between 1 and U on the optimal solution value is partitioned into u intervals

$$[1,(1+\varepsilon)^{\frac{1}{n}}[,\ [(1+\varepsilon)^{\frac{1}{n}},(1+\varepsilon)^{\frac{2}{n}}[,\ [(1+\varepsilon)^{\frac{2}{n}},(1+\varepsilon)^{\frac{3}{n}}[,\ \ldots,[(1+\varepsilon)^{\frac{u-1}{n}},(1+\varepsilon)^{\frac{u}{n}}[$$

with $u := \lceil n \log_{1+\varepsilon} U \rceil$. Note that u is of order $O(1/\varepsilon \cdot n \log U)$ and hence polynomial in the length of the encoded input. The main point of this construction is to guarantee that in every interval the upper endpoint is exactly $(1+\varepsilon)^{\frac{1}{n}}$ times the lower endpoint.

To achieve an *FPTAS* the dynamic programming algorithm from Section 13.1.2 is adapted to the partitioned profit space. Instead of considering function y for all integer profit values, only the value 0 and the u lower endpoints of intervals as possible profit values are considered. The definition of the resulting dynamic programming function $\tilde{y}$ is slightly different from y. An entry $\tilde{y}(\tilde{v}) = w$, where $\tilde{v}$ is the lower endpoint of an interval of the partitioned profit space, indicates that there exists a subset of items with weight w and a profit of *at least* $\tilde{v}$.

The update operation where item j is added to every entry $\tilde{y}(\tilde{v})$ is modified in the following way. We compute the profit attained from adding item j to the current entry. The resulting value $\tilde{v} + p_j$ is *rounded down* to the nearest lower endpoint of an interval $\tilde{u}$. The weight in the corresponding dynamic programming function entry is compared to $\tilde{y}(\tilde{v}) + w_j$. The minimum of the two values is stored as function value $\tilde{y}(\tilde{u})$.

The running time of the approach in [132] is bounded by $O(nu)$ and hence by $O(1/\varepsilon \cdot n^2 \log U)$. The space requirement is $O(n+u)$, i.e. $O(n \log U \cdot 1/\varepsilon)$, after avoiding the explicit storage of subsets for every dynamic programming entry (see Section 3.3).

Theorem 13.1.3 *The above algorithm computes a* $(1-\varepsilon)$*-approximation for every reachable profit value of a knapsack problem.*

Proof. The correctness of the statement is shown by induction over the set of items. In particular, we will show the following claim.

Claim: After performing all update operations with items $1,\ldots,j$ for some $j \in \{1,\ldots,n\}$ for the optimal function y and the approximate function $\tilde{y}$ there exists for every entry $y(v)$ an entry $\tilde{y}(\tilde{v})$ with

$$\text{(i)} \quad \tilde{y}(\tilde{v}) \leq y(v) \quad \text{and} \qquad \text{(ii)} \quad (1+\varepsilon)^{\frac{j}{n}} \tilde{v} \geq v.$$

Evaluating (ii) for $j = n$ immediately yields with $\frac{1}{1+\varepsilon} \geq 1 - \varepsilon$ the desired result.

To prove the Claim for $j = 1$ we add item 1 into the empty knapsack. Hence we get an update of the optimal function with $y(p_1) = w_1$ and of the approximate function with $\tilde{y}(\tilde{v}) = w_1$ where $\tilde{v}$ is the largest interval endpoint not exceeding p_1. Property (i) holds with equality, whereas (ii) follows from the fact that p_1 and $\tilde{v}$ are in the same interval and hence $(1+\varepsilon)^{\frac{1}{n}} \tilde{v} \geq p_1$.

Assuming the Claim to be true for $j-1$ we can show the properties for j by investigating the situation after trying to add item j to every function entry. Clearly, those entries of y which were not updated by item j fulfill the claim by the induction hypothesis since (ii) holds for $j-1$ and even more so for j.

Now let us consider an entry of y which was actually updated, i.e. item j was added to $y(v)$ yielding $y(v+p_j)$. Since the Claim is true before considering item j, there exists $\tilde{y}(\tilde{v})$ with $(1+\varepsilon)^{\frac{j-1}{n}} \tilde{v} \geq v$, i.e., $\tilde{v} \geq v/(1+\varepsilon)^{\frac{j-1}{n}}$.

In the *FPTAS* the value $\tilde{v}+p_j$ is computed and rounded down to some lower interval endpoint $\tilde{u}$. From the interval construction we have $(1+\varepsilon)^{\frac{1}{n}} \tilde{u} \geq \tilde{v}+p_j$. Putting things together this yields

$$(1+\varepsilon)^{\frac{1}{n}} \tilde{u} \geq v/(1+\varepsilon)^{\frac{j-1}{n}} + p_j \geq (v+p_j)/(1+\varepsilon)^{\frac{j-1}{n}}.$$

Rearranging terms, this proves that $\tilde{u}$ satisfies (ii) for $v+p_j$. Property (i) follows immediately by applying the claim for $y(v)$ since

$$\tilde{y}(\tilde{u}) = \min\{\tilde{y}(\tilde{u}), \tilde{y}(\tilde{v}) + w_j\} \leq y(v) + w_j = y(v+p_j).$$

□

The main point of developing the above *FPTAS* from [132] was the approximation of every reachable profit value and not only the optimal solution value, since this will allow us to extend the *FPTAS* to the multiobjective case. In itself, this new *FPTAS* for (KP) does not improve on the previously best known *FPTAS* presented in Section 6.2.

The extension of this *FPTAS* to the multiobjective problem can be performed in a completely analogous way as for the exact dynamic programming scheme of

Section 13.1.2. The partitioning of the profit space is done for every dimension, which yields $u_k := \lceil n \log_{1+\varepsilon} U_k \rceil$ intervals for every objective k. Instead of adding an item j to all lower interval endpoints $\tilde{v}$ as in the one-dimensional case we now have to add it to every possible t-tuple $(\tilde{v}_1, \ldots, \tilde{v}_t)$, where $\tilde{v}_k$ is either 0 or a lower interval endpoint of the k-th profit space partitioning. The resulting objective values $\tilde{v}_1 + p_{1j}, \ldots, \tilde{v}_t + p_{tj}$ are all rounded down for every objective to the nearest interval endpoint. At the resulting t-tuple of lower interval endpoints a comparison to the previous dynamic programming function entry is performed.

Since the "projection" of this algorithm to any single objective yields exactly the *FPTAS* for (KP) as discussed above, the correctness of this method follows from Theorem 13.1.3. The running time of this approach is clearly bounded by $O(n \prod_{k=1}^{t} u_k)$ and hence in $O(n(1/\varepsilon \cdot n \log U_{\max})^t)$. Summarizing, the following statement of [132] has been shown.

Theorem 13.1.4 *For every constant t, there is an* FPTAS *for the multiobjective knapsack problem with t objectives whose running-time is bounded by* $O(n^{t+1} \cdot (1/\varepsilon)^t \cdot (\log U_{\max})^t)$.

A comparison with the running-times of the *FPTAS* following from the existence results by Safer and Orlin [414] resp. Papadimitriou and Yannakakis [368] is somewhat unfair since their goal was not to develop a particular efficient algorithm for a specific problem. However, for the sake of illustration we mention the running time bounds that are obtained by a straightforward application of their constructions to the knapsack problem. Without giving any details, we find that the running-time of an *FPTAS* for (MOKP) following from Theorem 11 in [414] can be bounded by $O(n^{4t+1} \cdot (1/\varepsilon)^{2t} \cdot (\log U_{\max})^t)$. For the *FPTAS* following from Theorem 4 in [368] we obtain a slightly worse running time of $O(n^{4t+1} \cdot (1/\varepsilon)^{3t} \cdot (\log U_{\max})^t)$.

13.1.5 A *PTAS* for (MOd-KP)

In this section, we present a *PTAS* for (MOd-KP). This *PTAS* is taken from the paper by Erlebach, Kellerer and Pferschy [132]. Recall from Theorem 9.4.1 that the existence of an *FPTAS* to the two-dimensional knapsack problem (2-KP), would imply $\mathcal{P} = \mathcal{NP}$. Hence, we can only hope to develop a *PTAS* for (MOd-KP). For the construction of this *PTAS* it is made use of some of the ideas from the *PTAS* for the d-dimensional knapsack problem(d-KP) due to Frieze and Clarke [156] (see Section 9.4.2).

Let $\varepsilon > 0$ be given. Recall that we assume that both d and t are constants. Choose $\delta > 0$ and $\mu > 0$ to be positive constants such that $(1+\delta)(1+\mu) \leq 1 - \varepsilon$. Using the algorithm of Frieze and Clarke the algorithm computes for each k, $1 \leq k \leq t$, an approximation U_k of (d-KP) with objective p_{k*} (i.e., with $p_j = p_{kj}$ for all items j). In particular, any feasible solution S for an instance I satisfies

$$0 \leq V_k(S) \leq (1+\delta)V_k(U_k)$$

for all $1 \leq k \leq t$. Therefore, the set of all objective vectors of all feasible solutions to I is contained in

$$[0,(1+\delta)V_1(U_1)] \times [0,(1+\delta)V_2(U_2)] \times \cdots \times [0,(1+\delta)V_t(U_t)].$$

Define $u_k := \lceil \log_{1+\delta} V_k(U_k) \rceil$. With respect to the k-th objective, we consider the following (lower) bounds for objective values:

- $u_k + 1$ lower bounds $\ell_{kr} = (1+\delta)^{r-1}$, for $1 \leq r \leq u_k + 1$, and
- the lower bound $\ell_{k0} = 0$.

The algorithm enumerates all tuples $(r_1, r_2, \ldots, r_{t-1})$ of $t-1$ integers satisfying $0 \leq r_k \leq u_k + 1$ for all $1 \leq k < t$. Note that the number of such tuples is polynomial in the size of the input if t and δ are constants. Roughly speaking, the algorithm considers, for each tuple $(r_1, r_2, \ldots, r_{t-1})$, the subspace of solutions S satisfying $V_k(S) \geq \ell_{kr_k}$ for all $1 \leq k < t$, and tries to maximize $V_t(S)$ among all such solutions.

For this purpose, the basic idea in [132] is to relax the problem and formulate it as a linear program containing a variable x_j, bounded by $0 \leq x_j \leq 1$, for each item j. An integral solution is then obtained from an optimal fractional solution by rounding down all fractional variables. In order to ensure that the rounding has only a very small effect on the values of the t objective functions, we must avoid the case that items contributing the largest profits to some objective are rounded down. Therefore, the algorithm "guesses" for each objective the h items contributing the largest profits to that objective. For these items, the corresponding variables x_j are fixed to 1. Then a fractional variable that is rounded down can decrease each objective function by at most a $1/h$-fraction. If h is chosen large enough, it can thus be ensured that the rounded solution is sufficiently close to the fractional solution with respect to all t objective functions. Of course, the "guessing" is implemented by complete enumeration, i.e., by trying all possibilities of choosing the h items of largest profit for each of the t objectives.

The detailed implementation of this idea is as follows. Let $h := \lceil (t+d-1)(1+\mu)/\mu \rceil$. Let $A_1, \ldots, A_t$ be subsets of $\{1, \ldots, n\}$ with cardinality at most h. The algorithm enumerates all possibilities for choosing such sets $A_1, \ldots, A_t$. Intuitively, the set A_k represents the selected items of highest profit with respect to p_{k*}. Set $A := A_1 \cup A_2 \cup \cdots \cup A_t$. For $1 \leq k \leq t$ and $p_k^{\min} := \min\{p_{kq} \mid q \in A_k\}$, let

$$F_k := \{j \in N \setminus A_k \mid p_{kj} > p_k^{\min}\}.$$

Intuitively, F_k is the set of all items that cannot be put into the knapsack if A_k are the items with highest profit with respect to p_{k*} in the solution. Let $F = F_1 \cup F_2 \cup \cdots \cup F_t$.

If $F \cap A \neq \emptyset$, the current choice of the sets A_k is not consistent, and the algorithm continues with the next possibility of choosing the sets $A_1, A_2, \ldots, A_t$.

So let us assume that $F \cap A = \emptyset$. Consider the following *linear programming relaxation*:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{n} p_{tj} x_j && (13.3)\\
\text{subject to} \quad & \sum_{j=1}^{n} w_{kj} x_j \leq c_k, \quad k = 1, \ldots, d,\\
& \sum_{j=1}^{n} p_{kj} x_j \geq \ell_{kr_k}, \quad k = 1, \ldots, t-1,\\
& x_j = 0, \quad j \in F,\\
& x_j = 1, \quad j \in A,\\
& 0 \leq x_j \leq 1, \quad j \in \{1, \ldots, n\} \setminus (F \cup A).
\end{aligned}$$

Let $\hat{x}$ be an optimal (basic feasible) solution vector for (13.3). Such a vector $\hat{x}$ can be computed in polynomial time, if it exists. (If no such vector exists, proceed to the next combination of sets $A_1, \ldots, A_t$.) As (13.3) has only $t + d - 1$ non-trivial constraints, at most $t + d - 1$ components of $\hat{x}$ are fractional. Now an integral vector $\bar{x}$ is obtained by rounding down $\hat{x}$, i.e., we set $\bar{x}_j := \lfloor \hat{x}_j \rfloor$. From the integral vector $\bar{x}$, we obtain a solution S to (MOd-KP) by letting $S = \{j \in \{1, 2, \ldots, n\} \mid \bar{x}_j = 1\}$.

For each possibility of choosing $A_1, \ldots, A_t$ consistently, we either find that (13.3) has no solution, or we compute an optimal fractional solution to (13.3) and obtain a rounded integral solution S. We output all integral solutions that are obtained in this way (if any). The procedure is repeated for every tuple $(r_1, r_2, \ldots, r_{t-1})$. This completes the description of the algorithm. If desired, the set of solutions output by the algorithm can be reduced by discarding solutions that are weakly dominated by other solutions in the set. The following theorem [132] shows that the above algorithm is indeed a *PTAS*.

Theorem 13.1.5 *For every pair of constants d and t, the above algorithm is a* PTAS *for (MOd-KP).*

Proof. We show that the algorithm described above is a $(1 - \varepsilon)$-approximation algorithm for (MOd-KP).

First, we argue that the running time is polynomial. Initially, the approximation algorithm for (d-KP) is called t times. This gives the solutions $U_1, \ldots, U_t$ and the numbers $u_1, \ldots, u_t$. Then the algorithm enumerates all tuples of $t - 1$ numbers $r_1, \ldots, r_{t-1}$ satisfying $0 \leq r_k \leq u_k + 1$ for all $1 \leq k \leq t - 1$. There are $O(u_1 u_2 \cdots u_{t-1})$ such tuples. As each u_k is polynomial in the size of the input, the total number of tuples is also bounded by a polynomial. For each tuple, all combinations of choosing

sets $A_1,\ldots,A_t$ of cardinality at most h are enumerated. There are $O(n^{t(t+m-1)(1+\mu)/\mu})$ such combinations. For each combination, the linear program (13.3) is created and solved in polynomial time. Since d, t, δ and μ are constants, the overall running-time is polynomial in the size of the input.

Now, we analyze the approximation ratio. Consider an arbitrary feasible solution G. We have to show that the output of the algorithm contains a solution S that is a $(1-\varepsilon)$-approximation of G. For $1 \leq k \leq t-1$, define $r_k := \max\{r \mid \ell_{kr} \leq V_k(G)\}$. It follows that $\ell_{kr_k} \leq V_k(G) \leq \ell_{kr_k}(1+\delta)$.

For $1 \leq k \leq t$, let A_k be a set that contains $\min\{h, |G|\}$ items in G with largest profit p_{k*}. When the algorithm considers the tuple $(r_1, r_2, \ldots, r_{t-1})$ and the sets $A_1,\ldots,A_t$, the linear program (13.3) is feasible, because G constitutes a feasible solution. Therefore, the algorithm obtains a fractional solution S_f and outputs a rounded integral solution S for this tuple $(r_1, r_2, \ldots, r_{t-1})$ and for this combination of sets $A_1,\ldots,A_t$. If $|G| \leq h$, the rounded solution S is at least as good as G with respect to all t objectives, because it contains G as a subset. Thus, the solution S output by the algorithm weakly dominates G in this case.

Now assume that $|G| > h$. Consider objective k. We have that $V_k(S_f) \geq V_k(G)/(1+\delta)$. For $1 \leq k < t$, this is because S_f is a feasible solution to (13.3). For $k = t$, we even have $V_t(S_f) \geq V_t(G)$, because S_f is an optimal fractional solution to (13.3).

Furthermore, we claim that $V_k(S) \geq V_k(S_f)/(1+\mu)$. Consider some objective k, $1 \leq k \leq t$. When S is obtained from S_f, at most $t+d-1$ items are lost, because S_f has at most $t+d-1$ fractional items, as we noted above. Let $\bar{p}$ be the smallest profit among the h items with highest profit in G with respect to objective k. We have $V_k(S_f) \geq h\bar{p}$ and $V_k(S) \geq V_k(S_f) - (t+d-1)\bar{p}$. Combining these two inequalities, we get

$$\begin{aligned} V_k(S) &\geq V_k(S_f) - \frac{t+d-1}{h} h\bar{p} \geq V_k(S_f) - \frac{t+d-1}{h} V_k(S_f) = \\ &= V_k(S_f)\left(1 - \frac{t+d-1}{h}\right) \geq V_k(S_f)\left(1 - \frac{\mu}{1+\mu}\right) = \\ &= \frac{V_k(S_f)}{1+\mu}. \end{aligned}$$

This results in

$$V_k(S) \geq \frac{V_k(S_f)}{1+\mu} \geq \frac{V_k(G)}{(1+\mu)(1+\delta)}.$$

Since we have chosen μ and δ such that $(1+\mu)(1+\delta) \leq 1/(1-\varepsilon)$, we obtain that S is a $(1-\varepsilon)$-approximation of G.

As the above argument is valid for any feasible solution G, we have shown that the set of solutions output by the presented algorithm is indeed a $(1-\varepsilon)$-approximation of the Pareto frontier. □

13.1.6 Metaheuristics

An enormous amount of papers applying metaheuristic algorithms to multiobjective combinatorial optimization problems has been published during the last ten years. Like in Section 9.5.5 we will restrict ourselves to mentioning some papers which directly apply metaheuristic methods to (MOKP). For a detailed survey we refer again to the book by Ehrgott and Gandibleux [128].

The *multiobjective tabu search method* by Gandibleux, Mezdaoui and Fréville [162] employs weighted sum scalarization (WSS) described in Section 13.1.2 in order to obtain the efficient frontier. The tabu memories are used for updating the weights to diversify the search in the objective space. This approach was applied by Gandibleux and Fréville [161] to the knapsack problem with two objectives, exploiting the fact that the decision space could be reduced taking into account the properties of (MOKP). Hansen [213] invented a similar approach: It works with a set of non-dominated solutions, each with its own tabu list. By manipulations on the weights these solutions are dispersed over the efficient frontier.

All simulated annealing methods are slight variations of the classical simulated annealing technique. Czyzak and Jaszkiewicz [96] developed *Pareto simulated annealing*. This procedure introduces a sample of so-called generating solutions. These solutions are simultaneously optimized in a way similar to classical simulated annealing. Ulungu et al. [463] introduce the so-called MOSA method: The method starts with a predefined weight vector. Then, simulated annealing is independently run for each weight vector. The total set of all obtained solutions is then filtered to get a good approximation for the efficient frontier. Teghem, Ulungu and Tuyttens [453] embedded the MOSA method in an interactive procedure. Another interactive method which uses both simulated annealing and tabu search was introduced by Alves and Climacao [10].

Many multiobjective evolutionary algorithms are based on the *vector evaluated genetic algorithm* developed by Schaffer [423]. It divides the population of starting solutions into equally sized subpopulations. For each subpopulation one of the objectives is optimized. A selection procedure is performed independently on the subpopulations while evolution is done on the whole set of populations.

Laumanns, Zitzler and Thiele [294] emphasized that the use of elite solutions could improve evolutionary multi-objective search significantly. This was confirmed by Gandibleux, Morita and Katoh [163] for generating the efficient frontier for the (MOKP). Zitzler and Thiele [502] performed a comparative study of four different multiple objective evolutionary algorithms with a new approach, the *strength Pareto evolutionary algorithm* (SPEA), on the basis of the (MOKP), with the result that (SPEA) outperforms the other four algorithms. Jaszkiewicz [248] compared (SPEA) with a multiple genetic local search algorithm on the same set of instances. He claims that this approach is able to generate much better approximations to the non-dominated sets than (SPEA). More results for the application of evolutionary

algorithms to (MOKP) can be found in the Ph.D. thesis by Zitzler [501]. We refer to the homepage by Coello Coello [87] for an updated list of references on evolutionary multiobjective optimization.

Finally, let us mention a hybrid heuristic for (MOKP) by Abdelaziz, Krichen and Chaouachi [2] which combines tabu search and genetic algorithms.

13.2 The Precedence Constraint Knapsack Problem (PCKP)

The *precedence constraint knapsack problem* (PCKP) or *partially ordered knapsack problem* is a generalization of (KP) which includes a partial order on the items. "Item i *precedes* item j" means that item j can only be packed into the knapsack if item i is also packed into the knapsack. For an instance I of (PCKP) the precedence constraints are modelled by a directed, noncyclic graph $G_I = (V_I, A_I)$. The vertex set $V_I = \{1,\ldots,n\}$ corresponds to the items. The existence of an arc $(i,j) \in A_I$ means that item i precedes item j. Formally, (PCKP) can be defined by adding condition (13.4) to the integer linear programming formulation (1.1) to (1.3) of (KP):

$$\begin{aligned} \text{(PCKP)} \qquad \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_i \geq x_j, \quad (i,j) \in A_I, \qquad (13.4) \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned}$$

Note that it makes sense to define (PCKP) also for negative profit values since due to the precedence constraints, sometimes items with negative profits must also be contained in an optimal solution. This case will be covered by the dynamic programming algorithms presented in this section.

There is a wide range of applications for (PCKP). Ibarra and Kim [242] mention an investment problem with weights as costs and profits in which certain investments can only be made if certain other ones have also be made. Stecke and Kim [444] consider tool management problems. Shaw, Cho and Chang [433] discuss applications in local access telecommunication network design.

Before we continue we recall some basic definitions from graph theory. For an introduction to basic graph theory we refer to standard textbooks like Jungnickel [256]. An undirected graph $G = (V,E)$ with vertex set V and edge set E is a *bipartite graph* if the vertex set V can be partitioned into two subsets V_1 and V_2 such each edge of E has one vertex in V_1 and the other one in V_2. A graph is called a *complete graph* if every pair of vertices forms an edge.

For a directed graph $G = (V,A)$ with vertex set V and arc set A an arc $a \in A$ between two vertices v_i and v_j consists of an ordered pairs of vertices (v_i, v_j), v_i being called the *initial vertex* of a and v_j the *terminal vertex* of a. Each arc is assigned an orientation indicated by an arrow that is drawn from the initial to the terminal vertex. A *directed path* in a directed graph G is a finite sequence of vertices $v_0, v_1, \ldots, v_\ell$ with *end vertices* v_0 and v_ℓ such that all (v_i, v_{i+1}) $(0 \leq i \leq \ell - 1)$ are arcs of G and all vertices in this path except the end vertices are distinct. A directed path in a tree is determined by the end vertices v_0 and v_ℓ and abbreviated by $P[v_0, v_\ell]$. A directed path is called *closed* or *directed circuit* if its end vertices are identical. A directed graph is said to be *acyclic* if it has no directed circuits. Note that any *partially ordered set* can represented by an acyclic directed graph.

If there is a directed path in an acyclic, directed graph from vertex v_i to v_j, then v_i is called *predecessor* of v_j and v_j *successor* of v_i, respectively.

An *out-tree* is a directed tree with a distinguished vertex called the *root* such that there is a directed path (which is unique) from the root to every vertex in the tree. The vertices which are terminal vertices of arcs with initial vertex v are the *children* of v. Let $s(v)$ denote the number of children of v. A vertex w which has a vertex v as its child is said to be the *parent* of v. Note that in an out-tree every vertex besides the root has exactly one parent By reversing the orientations of the arcs of an out-tree one obtains an *in-tree*. Applying *depth-first search* to an out-tree means that arcs are explored, starting with the root, out of the most recently found vertex v that still has unexplored arcs leaving it. When all arcs leaving v have been explored, the search backtracks to explore arcs leaving the vertex from which v was reached.

Johnson and Niemi [252] were the first to consider (PCKP) thoroughly. They showed that (PCKP) is strongly $\mathcal{NP}$-hard. Thus, there is no hope to find a pseudopolynomial algorithm or an *FPTAS* for (PCKP) unless $\mathcal{P} = \mathcal{NP}$. For the proof of the following theorem we use the famous $\mathcal{NP}$-complete clique problem (see Garey and Johnson [164, GT19]):

Clique problem CLIQUE
Instance: An undirected graph $G = (V,E)$ with vertex set V and edge set E and an integer $K \leq |V|$
Question: Is there a complete subgraph of G with K vertices?

Theorem 13.2.1 *Problem (PCKP) is $\mathcal{NP}$-hard in the strong sense even if the profits are equal to the weights and the underlying graph G is bipartite.*

Proof. The proof is performed by reduction from CLIQUE. Given an instance of CLIQUE we define an instance I of (PCKP) with capacity c as follows:

$$\begin{aligned} V_I &= V \cup E, \\ A_I &= \{(v,e) \mid v \in V, e \in E, v \text{ is vertex of } e\}, \\ w(v) &= p(v) = |E| + 1, \quad v \in V, \end{aligned}$$

$$w(e) = p(e) = 1, \quad e \in E,$$
$$c = K(|E|+1) + \binom{K}{2}.$$

We show that there is a solution for an instance I of (PCKP) with profit at least c if and only if G has a complete subgraph with K vertices. First, assume there is a complete subgraph $G' = (V', E')$ of G with $|V'| = K$. From $|E'| = K(K-1)/2$ follows immediately that the items corresponding to $V' \cup E'$ form a solution with profit c.

Then assume that there is a solution for I with profit at least c. Since the profits are equal to the weights and c is the knapsack capacity, the desired solution $V_I' \subseteq V_I$ has total profit exactly c. This implies $|V_I' \cap V| = K$ and $|V_I' \cap E| = \binom{K}{2}$ which proves that the induced subgraph of V_I' is complete. □

13.2.1 Dynamic Programming Algorithms for Trees

In this section we will present two dynamic programming algorithms to solve (PCKP) problems for out-trees exactly in pseudopolynomial running time.

The first dynamic programming algorithm for (PCKP) for out-trees is due to Johnson and Niemi [252]. They call their algorithm *left-right approach*. It can be described as follows.

Denote the underlying out-tree by $T = T(V,A)$ with $V = \{1,\ldots,n\}$ and compute an upper bound U for the objective function value, e.g. by using the solution of the LP-relaxation from Section 2.2. W.l.o.g. assume the vertices $1,\ldots,n$ of the out-tree T to be enumerated in depth-first order. Note that vertex 1 corresponds to the root. Let $T(j,i)$, $j = 1,\ldots,n$, $i = 0,\ldots,s(j)$, denote the subtree of T induced by vertex j, the first i children of j and all their successors, and all vertices of V with indices lower than that of j. Thus, there is some k with $j \leq k \leq n$ such that the vertex set of $T(j,i)$ is equal to $N_k := \{1,\ldots,k\}$. Note that the same set N_k may correspond to different $T(j,i)$. As an example $T(1,2)$ and $T(4,2)$ both represent the tree depicted in Figure 13.1.

A *solution for* $T(j,i)$ is a set $V_1 \subseteq N_k$ with $j \in V_1$, which contains for each $\ell \in V_1$ all predecessors of ℓ in V and has total weight at most c. For a triple $(j,i,\bar{p})$ $(0 \leq \bar{p} \leq U)$ set

$$y(j,i,\bar{p}) := \min\{w(V_1) \mid V_1 \text{ is a solution for } T(j,i) \text{ and } p(V_1) \geq \bar{p}\} \cup \{\infty\}.$$

Thus, $y(j,i,\bar{p})$ denotes the smallest weight for which a solution of $T(j,i)$ with profit at least $\bar{p}$ exists. Note that the optimum solution value z^* is the maximum value $\bar{p}$ for which $y(1,s(1),\bar{p}) < \infty$.

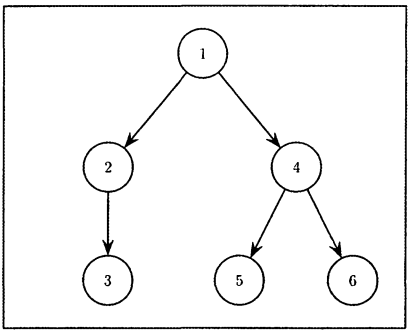


Fig. 13.1. A simple out-tree.

The main idea of the algorithm is a suitable ordering of the subtrees $T(j,i)$ which corresponds to the structure of the dynamic programming recursion. Starting with $T(1,0)$ the values $y(j,i,\bar{p})$ are calculated by expanding the subtrees first down the left arc of the tree and then to the right. This means, $y(j,i,\bar{p})$ is determined before $y(j,i+1,\bar{p})$ ($j \in V, 0 \leq i < s(j)$), and if j_i is the i-th child of j, $y(j,i-1,\bar{p})$ is determined before $y(j_i,0,\bar{p})$ and $y(j_i,0,\bar{p})$ is determined before $y(j,i,\bar{p})$. The recursion of the left-right approach is computed according to the following three rules:

1. $i = 0$ and $j = 1$:
$$y(1,0,\bar{p}) = \begin{cases} w_1 & \text{if } p_1 \geq \bar{p}, \\ \infty & \text{otherwise.} \end{cases}$$

2. $i = 0$ and $j > 1$:
$$y(j,0,\bar{p}) = \begin{cases} y(u,t-1,r) + w_j & \text{if } y(u,t-1,r) + w_j \leq c, \\ \infty & \text{otherwise.} \end{cases}$$
where j is the t-th child of u and $r = \max\{0, \bar{p} - p_j\}$.

3. $1 \leq i \leq s(j)$:
$$y(j,i,\bar{p}) = \min\{y(j,i-1,\bar{p}), y(j_i,s(j),\bar{p})\}$$

The initialization of the left-right approach for the root 1 is done in Step 1. Note that the vertex set of $T(j,0)$ is equal to N_j and the vertex set of $T(u,t-1)$ is equal to N_{j-1}. By definition item j must be contained in the solution for $T(j,0)$. Thus, Step 2 has only to determine whether the total weight of $y(u,t-1,r) + w_j$ does exceed c or not. The classical knapsack recursion in Step 3 is executed by distinguishing whether j_i, the i-th child of j, is put into the knapsack or not. In this step, the evaluated trees $T(j,i)$ and $T(j_i,s(j))$ have the same vertex set which consists of N_{j_i} and all successors of j_i, whereas the tree $T(j,i-1)$ has vertex set N_{j_i-1}.

Example: Consider the tree with six vertices depicted in Figure 13.1. The algorithm computes the values $T(j,i)$ in the following order: $T(1,0), T(2,0), T(3,0), T(2,1), T(1,1), T(4,0), T(5,0), T(4,1), T(6,0), T(4,2), T(1,2)$. □

Since the computation of the optimal solution vector can be done using standard techniques from Section 2.3, we omit a description. The running time of the left-right approach is in $O(nU)$. Note that the left-right approach can easily be adapted for in-trees and converted into an *FPTAS* for (PCKP) with out-trees by an appropriate scaling of the profit values as in Section 2.6.

A variant of the left-right approach was used by Cho and Shaw [80] for solving (PCKP) with out-trees. Instead of dynamic programming by profits Cho and Shaw perform dynamic programming by weights which results in an algorithm with running time in $O(nc)$. Again the vertices $1,\ldots,n$ of the out-tree T are going to be enumerated in depth-first order. Let a solution for a subtree $T(j,i)$ be defined as above. Then, for $(j,i,\bar{c})$ $(0 \leq \bar{c} \leq c)$ define

$$z(j,i,\bar{c}) := \max\{p(V_1) \mid V_1 \text{ is a solution for } T(j,i) \text{ and } w(V_1) \leq \bar{c}\}. \tag{13.5}$$

Now $z(j,i,\bar{c})$ denotes the largest profit for which a solution of $T(j,i)$ with weight at most $\bar{c}$ exists. Note that $z^* = z(1,s(1),c)$.

The recursion relations are analogous:

1. $i = 0$ and $j = 1$:
$$z(1,0,\bar{c}) = \begin{cases} p_1 & \text{if } w_1 \leq \bar{c}, \\ -\infty & \text{otherwise.} \end{cases}$$

2. $i = 0$ and $j > 1$:
$$z(j,0,\bar{c}) = \begin{cases} z(u,t-1,\bar{c}-w_j) + p_j & \text{if } \sum\limits_{\ell \in P[1,j]} w_\ell \leq \bar{c}, \\ -\infty & \text{otherwise.} \end{cases}$$
where j is the t-th child of u.

3. $1 \leq i \leq s(j)$:
$$z(j,i,\bar{c}) = \max\{z(j,i-1,\bar{c}), z(j_i,s(j),\bar{c})\}$$

13.2.2 Other Results for (PCKP)

By Theorem 13.2.1 we know that there are no pseudopolynomial algorithms for (PCKP) with general graphs. Samphaiboon and Yamada [417] present a simple dynamic programming algorithm with exponential running time which partitions the solution set by distinguishing whether an item j is contained in the optimal solution or not.

W.l.o.g. assume the vertices of G_I to be *topologically sorted*, i.e. $(i,j) \in A_I$ means that $i < j$. Let $\hat{G} = (\hat{V}, \hat{A})$ be a subgraph of G_I and denote by α the vertex with smallest index in $\hat{V}$ and by $S(\alpha)$ the set of all successors of α in $\hat{G}$. Let $PCKP(\hat{G}, \bar{c})$ be the (PCKP) for a subgraph $\hat{G}$ and capacity $\bar{c}$ ($0 \leq \bar{c} \leq c$). Analogously to Section 2.3, the optimal solution value for $PCKP(\hat{G}, \bar{c})$ is denoted by $z(\hat{G}, \bar{c})$. We set $x(\hat{G}, \bar{c}) = 1$ if α is in the optimal solution set $X(\hat{G}, \bar{c})$ for $PCKP(\hat{G}, \bar{c})$ and 0 otherwise.

Depending on whether α is contained in the optimal solution set for the problem $PCKP(\hat{G}, \bar{c})$ or not, $PCKP(\hat{G}, \bar{c})$ is divided in two new subproblems. If $\alpha \in X(\hat{G}, \bar{c})$, we consider the subgraph G_L of $\hat{G}$ with vertices $\hat{V} \setminus \{\alpha\}$. Thus, the optimal solution value for $PCKP(\hat{G}, \bar{c})$ is p_α plus the optimal solution value for $PCKP(G_L, \bar{c} - w_\alpha)$. If $\alpha \notin X(\hat{G}, \bar{c})$, all successors of α have to be removed and we consider the subgraph G_R of $\hat{G}$ with vertices $\hat{V} \setminus \{S(\alpha)\}$. Then, the optimal solution value for $PCKP(\hat{G}, \bar{c})$ is the optimal solution value for $PCKP(G_R, \bar{c})$.

Thus, (PCKP) can be solved recursively, calculating the optimal solution values by

$$z(\hat{G}, \bar{c}) = \begin{cases} \max\{p_\alpha + z(G_L, \bar{c} - w_\alpha), z(G_R, \bar{c})\} & \text{if } \bar{c} \geq w_\alpha, \\ z(G_R, \bar{c}) & \text{otherwise,} \end{cases}$$

and the optimal solution sets by

$$x(\hat{G}, \bar{c}) = \begin{cases} 1 & \text{if } p_\alpha + z(G_L, \bar{c} - w_\alpha) \geq z(G_R, \bar{c}), \\ 0 & \text{otherwise.} \end{cases}$$

Noting that for $\hat{V}$ consisting of a single vertex α we have

$$z(\hat{G}, \bar{c}) = \begin{cases} p_\alpha & \text{if } w_\alpha \leq \bar{c} \quad \text{and} \quad p_\alpha \geq 0, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$x(\hat{G}, \bar{c}) = \begin{cases} 1 & \text{if } w_\alpha \leq \bar{c} \quad \text{and} \quad p_\alpha \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Samphaiboon and Yamada [417] propose a preprocessing algorithm, consisting of a greedy-type algorithm, to calculate a lower bound for the problem. They announce

that their dynamic programming together with the preprocessing method is able to solve (PCKP) with up to 2000 items in a few minutes of CPU time.

Shaw and Cho [432] propose a branch-and-bound algorithm for (PCKP) with out-trees. In the classical (KP) the split item s is used to determine an upper bound for the optimal solution value. Shaw and Cho extend the notion of a split item to (PCKP) with trees. The generalized split item can be found in $O(n^2)$ time and is used to calculate upper bounds in the branch-and-bound procedure. The branching is done through item deletion in contrast to item addition in the classical (KP). This branch-and-bound procedure experimentally outperforms the dynamic programming procedure presented by the same authors.

Woeginger [486] and Kolliopoulos and Steiner [284] explore (PCKP) with special graph classes different from trees. The tools for presenting their investigations in detail are too specialized for the scope of this book, so we just mention their main results without defining the underlying partial orders. For the latter we refer to the book by Brandstädt, Le and Spinrad [46] which gives a comprehensive survey on results for different kinds of graph classes, especially partially ordered sets.

Woeginger [486] gives pseudopolynomial algorithms for interval orders or bipartite convex orders. He also pointed out the relationship between (PCKP) and the one machine scheduling problem with precedence constraints and the objective to minimize the weighted sum of completion times. Using the combinatorial theory for partial orders Kolliopoulos and Steiner [284] provide a bicriteria *FPTAS* for (PCKP) with a 2-dimensional partial order. They also give a polynomial algorithm for the problem class where either $p_j = 0$ or $w_j = 0$ holds for all items j ($j = 1, \ldots, n$) and the graph G_I is bipartite and its bipartite complement is chordal bipartite. Note that this class does not include (KP) as a special case.

Finally, we want to mention two papers which study polyhedral problems for (PCKP). Boyd [44] investigates the polyhedral structure of the convex hull of feasible integer points when the precedence constraints are complicated by an additional constraint. A general procedure for inducing facets called rooting is introduced. Van de Leensel, van Hoesel and van de Klundert [467] discuss the complexity of obtaining classes of valid inequalities which are facet defining for (PCKP) using the sequential lifting procedure. We refer to Section 3.10 for an introduction to the knapsack polytope.

13.3 Further Variants

There are many other variants of the knapsack problem not mentioned in this chapter so far which appear in the literature. So, we had to make a more or less subjective selection of what to include. We will give a short description and summary of the results for the following problems in this section: the nonlinear knapsack problem,

the max-min knapsack problem, the minimization knapsack problem, the equality knapsack problem, the strongly correlated knapsack problem, the change-making problem, the collapsing knapsack problem, the parametric knapsack problem, the fractional knapsack problem, the set-union knapsack problem and finally the multi-period knapsack problem. Note that the on-line knapsack problem will be discussed in Section 14.8 in the context of stochastic analysis.

13.3.1 Nonlinear Knapsack Problems

In its most general form nonlinear knapsack problems can be described as solutions of the following problem:

$$\begin{aligned} &\text{maximize } f(x) \\ &\text{subject to } g(x) \leq c, \\ &\qquad\qquad x \in D \end{aligned}$$

where $x = (x_1, \ldots, n_n) \in \mathbb{R}^n$, $f(x)$ and $g(x)$ are continuous and differentiable functions, and $D \subseteq \mathbb{R}^n$. There is a huge variety of problems which can constructed based on this general definition. Most of them are continuous optimization problems. So we omit their description in a book which is mainly devoted to combinatorial optimization problems. For a detailed survey on general nonlinear knapsack problems we refer to the paper by Bretthauer and Shetty [48].

We only consider here the convex, separable, integer case and define a nonlinear knapsack problem (NLK) as follows:

$$\begin{aligned} \text{(NLK)} \qquad &\text{maximize } \sum_{j=1}^{n} f_j(x_j) \\ &\text{subject to } \sum_{j=1}^{n} g_j(x_j) \leq c, \\ &\qquad 0 \leq x_j \leq b_j,\ x_j \text{ integer}, \quad j = 1, \ldots, n. \end{aligned}$$

We assume b_j, c are positive integers, the functions $f_j(x_j)$ are concave and increasing, and the functions $g_j(x_j)$ are convex and decreasing ($j = 1, \ldots, n$). The (NLK) is also called the *nonlinear resource allocation problem*. There are many applications for (NLK) including the capital budgeting problem and production planning problems. Mathur, Salkin and Mohanty [339] showed how to convert (NLK) into a classical (KP) by using a piecewise linear approximation of the functions f_j and g_j. The corresponding piecewise linear function $h_j(x_j)$ is defined by the breakpoints $\{(g_j(i), f_j(i)) \mid i = 0, \ldots, b_j\}$. Then, (NLK) is equivalent to

$$\text{maximize} \quad \sum_{j=1}^{n} h_j(y_j)$$

$$\text{subject to} \quad \sum_{j=1}^{n} y_j \leq c,$$

$$y_j \in B_j := \{g_j(i) \mid i = 0,\ldots,b_j\}, \quad j = 1,\ldots,n.$$

The functions $h_j(y_j)$ are increasing concave functions for $g_j(0) \leq y_j \leq g_j(b_j)$ since the slope

$$s_{ij} := \frac{f_j(i) - f_j(i-1)}{g_j(i) - g_j(i-1)}$$

is decreasing in i.

Replacing each variable x_j by the sum of variables $\sum_{i=1}^{b_j} x_{ij}$ is equivalent to $x_j = i$, where

$$x_{1j} =,\ldots,= x_{ij} = 1, x_{i+1,j} =,\ldots,= x_{b_j,j} = 0.$$

By introducing the generalized weights $w_{ij} := g_j(i) - g_j(i-1)$ and the generalized profits $p_{ij} := f_j(i) - f_j(i-1)$ we get the program (NLK1).

(NLK1)

$$\text{maximize} \quad \sum_{j=1}^{n} \sum_{i=1}^{b_j} p_{ij} x_{ij} + \sum_{j=1}^{n} f_j(0)$$

$$\text{subject to} \quad \sum_{j=1}^{n} \sum_{i=1}^{b_j} w_{ij} x_{ij} \leq c - \sum_{j=1}^{n} g_j(0),$$

$$x_{ij} \in \{0,1\}, \quad i = 1,\ldots,b_j, \quad j = 1,\ldots,n.$$

From above we know that the function $h_j(y_j)$ is concave. Thus, if for an optimal solution of (NLK1) $x_{ij} = 1$ for some i, then $x_{kj} = 1$ holds for all $k < i$ and (NLK1) is really equivalent to (NLK). If the values w_{ij} and p_{ij} are all integers, (NLK1) is a classical (KP) and can be solved by dynamic programming in $O((\sum_{j=1}^{n} b_j)c)$ time.

Hochbaum [232] adapted the *FPTAS* of Lawler [295] for (KP) to construct an *FPTAS* for (NLK) with total computation time in

$$O(\frac{1}{\varepsilon}(n \log c + \log(\frac{1}{\varepsilon}) \log n + (\frac{1}{\varepsilon^2}) \log(\frac{1}{\varepsilon}))).$$

Independently, Kovalyov [289] developed an ε-approximation algorithm for (NLK) where the functions f_j and g_j need not to be concave or convex. Let the numbers L and U denote lower and upper bounds for the optimal solution value. Then, his algorithm has a running time in $O(nU \sum_{j=1}^{n} \min\{nU/(\varepsilon L), b_j\}/(\varepsilon L))$.

Bretthauer and Shetty [47] use a general framework for the continuous relaxation of (NLK) to get the so-called *multiplier search branch-and-bound algorithm* for solving (NLK). Heuristics are also given that round the continuous solution to obtain a feasible solution for (NLK). In [49] they present a "pegging algorithm" for solving the continuous problem and embed this pegging algorithm in a branch-and-bound procedure. Pegging algorithms exploit the fact that after ignoring some bounds on the variables the resulting relaxed problem is much easier to solve.

13.3.2 The Max-Min Knapsack Problem

Similarly to the multiobjective knapsack problem the *max-min knapsack problem* (MMKP) is obtained from the classical knapsack (KP) problem by introducing t profit values instead of one for every item. More precisely, an instance I of (MMKP) consists of a set of items $N = \{1,\dots,n\}$ and a knapsack with capacity c. Each item j ($j = 1,\dots,n$) has t profits p_{kj} ($k = 1,\dots,t$) and weight w_j ($j = 1,\dots,n$). With these t profits, t different objective functions are defined. The problem is to select items with total weight at most c such that the minimum total profit of the selected items over all objectives is maximized:

$$\begin{aligned} \text{(MMKP)} \qquad & \max \min_{k=1,\dots,t} \sum_{j=1}^{n} p_{kj} x_j \\ & \text{subject to} \sum_{j=1}^{n} w_j x_j \le c, \\ & x_j \in \{0,1\}, \quad j = 1,\dots,n. \end{aligned}$$

(MMKP) was first considered by Yu [494]. He mentioned applications in the area of robust optimization under uncertainty, e.g. the capital budgeting problem. The possible returns of an investment decision j depend on which out of t different scenarios is realized in the future, while the available budget c is known in advance. That investment should be made which maximizes the lowest return under all scenarios within the given budget.

By reduction from the set covering problem (see [164, SP5]) Yu showed that for t part of the input (MMKP) is strongly $\mathcal{NP}$-hard even if all weights are equal to one. If t is a constant, (MMKP) can still be solved in pseudopolynomial time with dynamic programming by weights. Define by $z_j(d, v_1,\dots,v_t)$ the optimal solution for the following integer linear program.

$$\max \min_{k=1,\dots,t} \sum_{i=1}^{j} (p_{ki} x_i + v_k)$$

$$\text{subject to} \quad \sum_{i=1}^{j} w_i x_i \leq d,$$
$$x_i \in \{0,1\}, \quad i = 1,\ldots,j.$$

Thus, $z_j(d,v_1,\ldots,v_t)$ is the optimal solution for item set $\{1,\ldots,j\}$, for a knapsack with capacity d and for increasing the k-th objective function by the value v_k, $1 \leq k \leq t$. Note that $z_n(c,0,\ldots,0)$ corresponds to the optimal solution value for (MMKP). Then, the recursions for $z_j(d,v_1,\ldots,v_t)$ are defined by

$$z_{j+1}(d,v_1,\ldots,v_t) =$$
$$= \begin{cases} z_j(d,v_1,\ldots,v_t) & \text{if } d < w_{j+1}, \\ \max\{z_j(d,v_1,\ldots,v_t), & \\ z_j(d-w_{j+1},v_1+p_{1(j+1)},\ldots,v_t+p_{t(j+1)})\} & \text{if } d \geq w_{j+1}. \end{cases}$$

Starting with the initialization

$$z_0(d,v_1,\ldots,v_t) = \min_{k=1,\ldots,t} v_k,$$

all values $z_j(d,v_1,\ldots,v_t)$, with $0 \leq v_k \leq \sum_{j=1}^{n} p_{jk}$ for $1 \leq k \leq t$, can be calculated in $O(ncP^t)$ where $P := \max_{k=1,\ldots,t} \{\sum_{j=1}^{n} p_{jk}\}$.

In the same paper [494] Yu uses a subgradient procedure to generate lower and upper bounds for (MMKP) based on surrogate relaxation. These surrogate lower and upper bounds are implemented in a branch-and-bound procedure. Iida [243] applies linear programming to obtain better lower and upper bounds and presents computational results for the corresponding branch-and-bound algorithm.

13.3.3 The Minimization Knapsack Problem

The minimization knapsack problem (MinKP) is a transformation of the traditional (KP) into a minimization problem. From a finite number of items a subset shall be selected with total weight at least c such that the total profit (cost) of the chosen items is minimized:

$$\text{(MinKP)} \qquad \text{minimize} \sum_{j=1}^{n} p_j y_j$$
$$\text{subject to} \sum_{j=1}^{n} w_j y_j \geq c,$$
$$y_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

Every (MinKP) can be formulated as a (KP) (and vice versa) by maximizing the profit of the items not contained in the knapsack under the condition that their total weight is at most $\sum_{j=1}^{n} w_j - c$. Consequently, if the binary variables $y_1, \ldots y_n$ form an optimal solution of the (MinKP), then the variables $x_j := 1 - y_j$ $(j = 1, \ldots, n)$ form an optimal solution of the corresponding (KP).

The only subject where (MinKP) and (KP) exhibit relevant differences are approximation algorithms. We know from Theorem 2.5.4 that Algorithm Ext-Greedy has a relative performance guarantee of 1/2. The corresponding version for (MinKP) is obtained by sorting the items in *increasing* order of profit weight ratio and packing the items into the knapsack until the total weight is not smaller than c. This solution is compared with the solution (if it exists) consisting of the single item with smallest profit and weight greater or equal than c. Unfortunately, this algorithm performs arbitrarily bad from a worst-case point of view. This can be easily seen by considering an instance with three items with profits $p_1 = 1$, $p_2 = c - 2$, $p_3 = 1$ and weights $w_1 = w_2 = c - 1$, $w_3 = 1$. The minimization version of algorithm Ext-Greedy will select items 1 and 2, while the optimum solution consists of items 1 and 3.

Gens and Levner [168] briefly outlined a multi-run greedy algorithm with performance guarantee 2, paving the road for a *FPTAS* using the techniques presented in Section 4.6. Güntzer and Jungnickel [201] found several variations of greedy-like algorithms which do not have the disadvantage of Ext-Greedy. They presented ε-approximation algorithms for (MinKP) which correspond to ε-approximation algorithms for (KP), as well.

13.3.4 The Equality Knapsack Problem

The *equality knapsack problem* (EKP) is a variant of the ordinary (KP) where strict equality should be satisfied in the capacity constraint. Hence, given n items with profits p_j and weights w_j, choose a subset of the items so that the total weight equals a given capacity c, and such that the total profit is maximized:

$$\begin{aligned} \text{(EKP)} \qquad & \text{maximize} \sum_{j=1}^{n} p_j x_j \\ & \text{subject to} \sum_{j=1}^{n} w_j x_j = c, \\ & x_j \in \{0,1\}, \quad j = 1, \ldots, n. \end{aligned}$$

Ram and Sarin [402] presented an exact algorithm based on branch-and-bound. Upper bounds are derived by solving the LP-relaxation of (EKP). The algorithm follows the same framework as for the ordinary (KP) described in Section 5.1: First,

the items are sorted according to decreasing efficiencies p_j/w_j and the greedy algorithm is used to find the split item s. Then a reduction algorithm is run as described in Section 5.1.3, also improving the lower bound as in equation (5.30). Finally, the reduced problem is solved by use of a best-first branch-and-bound algorithm.

Volgenant and Marsman [476] improved this algorithm by solving a core problem. A core is chosen as in Section 5.4, but to ensure feasibility of the core problem, each core problem is first solved by a heuristic. If the heuristic fails to find a feasible solution, a larger core is tried. If a feasible solution has been found, a reduction algorithm is applied followed by the branch-and-bound algorithm of Ram and Sarin. If optimality of the core solution can be proved, the algorithm terminates, otherwise it extends the core and repeats the above steps.

13.3.5 The Strongly Correlated Knapsack Problem

The subset sum problem may be seen as a special instance of the knapsack problem, but it is an equally relevant problem studied as a general problem. In a similar way, one may consider *strongly correlated knapsack problems* as a family of instances of (KP), or as a general model. Following the latter direction, we may define the problem as follows:

$$\text{(SCKP)} \qquad \text{maximize} \sum_{j=1}^{n}(w_j+K)x_j \tag{13.6}$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_j \le c, \tag{13.7}$$

$$x_j \in \{0,1\}, \quad j=1,\ldots,n. \tag{13.8}$$

where p_j are the profits, w_j are the weights, and $K \ge 0$ is a given constant.

Despite the fact that (SCKP) is one of the favorite benchmark instances for (KP), it has been complemented by relatively little theoretical work. Notice that (SCKP) contains (SSP) as a special case for $k = 0$ and hence the problem is $\mathcal{NP}$-hard.

Assume that the items are sorted according to increasing weights. Thus, the split item s is given as before by $s = \min\{j \mid \sum_{i=1}^{j} w_i > c\}$. As in (5.15) we may hence impose the *cardinality constraint*

$$\sum_{j=1}^{n} x_j \le s-1. \tag{13.9}$$

Pisinger [384] surrogate relaxed the constraints (13.7) and (13.9) with multipliers $\mu_1 = 1$ and $\mu_2 = K$, getting the problem

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n}(w_j+K)x_j \\ \text{subject to} \quad & \sum_{j=1}^{n}(w_j+K)x_j \le c+K(s-1) \\ & x_j \in \{0,1\}, \quad j=1,\ldots,n. \end{aligned} \tag{13.10}$$

The relaxed problem is an ordinary subset sum problem, and one immediately gets the upper bound of $c+K(s-1)$ for (SCKP). This choice of surrogate multiplier is optimal if just $\sum_{j=1}^{s} w_j \le \sum_{j=n-s+2}^{n} w_j$ with the above ordering.

In [384] it is shown that if the cardinality $\sum_{j=1}^{n} x_j^*$ of an optimal solution x^* to (13.10) is $s-1$ then the same solution is also optimal to (SCKP). Hence, an exact algorithm is presented which solves (13.10) with fixed cardinality k for $k := s-1, s-2, \ldots, 1$. Each of these problems can be solved by the dynamic programming algorithm described in Section 9.7. Various upper and lower bounds are derived to show that only a few values of k need to be considered for most instances. In particular, the larger K becomes, the fewer iterations are needed. Computational experiments confirm the benefits of this strategy, since most instances from the literature can be solved in milliseconds.

13.3.6 The Change-Making Problem

The *change-making problem* asks to reach a given *target sum* c using an unbounded amount of each weight w_i, where the set of available weights $w_1, \ldots, w_n$ frequently are denoted as the *denominations of coins*. The aim is to pay a fixed amount c with a minimum number of coins. The problem may be recognized as a minimization version of the unbounded knapsack problem with equality constraints, where all profits are 1. Formally it may be stated as follows:

$$\begin{aligned} \text{(CMP)} \qquad \text{minimize} \quad & \sum_{j=1}^{n} x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j = c, \\ & x_j \ge 0, \text{ integer}, \quad j=1,\ldots,n. \end{aligned}$$

The problem is $\mathcal{NP}$-hard (Lueker [309]) since the capacity constraint contains the SSP-DECISION problem as a special case (see Section A.2). This also means, that a feasible solution to the optimization problem does not necessarily exist, if the weights form an unsuitable combination.

Martello and Toth [326] present several lower bounds for (CMP), which can be used in designing an exact branch-and-bound algorithm. Chang and Gill [78] discuss sufficient properties of the weights $w_1, \ldots, w_n$ so that a greedy algorithm always will return an optimal solution. The problem lends itself well to dynamic programming as noticed by several authors. For a more detailed presentation of the change-making problem we refer to the book by Martello and Toth [335, ch. 5].

13.3.7 The Collapsing Knapsack Problem

The fact that the right-hand side of (1.2), i.e. the capacity value c, is a constant input value for (KP) may not apply to all applications of the knapsack problem. Instead, the amount of available resource may depend in some way on the solution structure. The most basic case of such a relationship is the dependence of c on the *number of packed items*. This means that the capacity is a functional $c(i) : \mathbb{N}_0 \to \mathbb{N}_0$ which represents a capacity value for every number of packed items. Thus the weight constraint (1.2) is written as

$$\sum_{j=1}^{n} w_j x_j \leq c\left(\sum_{j=1}^{n} x_j\right). \tag{13.11}$$

In most cases of a dependence between capacity and knapsack filling it makes sense to assume that $c(i)$ is a monotonous function. If $c(i)$ is a *decreasing* function the resulting problem is known as the *collapsing knapsack problem* (CKP). In this case the capacity will decrease the more items are packed indicating a sort of "overhead costs". Note that if $c(i)$ is a linear function then (CKP) is equivalent to (KP) since the ascent of the function can be added to the item weights.

The collapsing knapsack problem was introduced by Posner and Guignard [397] in 1978 and has applications in satellite communication where transmissions on the band require gaps between the portions of the band assigned to each user. Other applications are time-sharing computer systems, where each process run on the system causes additional overhead, and the design of a shopping center where both the type and size of a shop must be selected [397].

If $c(i)$ is an *increasing* function the resulting problem is defined as the *expanding knapsack problem* as mentioned in Pferschy, Pisinger and Woeginger [376]. It can be interpreted as a "rubber knapsack" whose capacity increases the more items it contains. Like the collapsing knapsack problem it has several applications, e.g. in budget control, where the trader gives discounts depending on the number of items purchased, or in manpower planning of state subsidized projects where the size of a grant depends on the number of employees.

Obviously, (KP) is a special case of (CKP) where $c(i)$ is a constant function. On the other hand, it was shown by Pferschy, Pisinger and Woeginger [376] that every instance of (CKP) can be written as an equivalent instance of (KP) with $2n$ items.

Consider the following reduction. Let $P := \sum_{j=1}^{n} p_j$ and $W := \sum_{j=1}^{n} w_j$. W.l.o.g. we may assume that $0 \leq c(i) \leq W$ for all i, as otherwise we replace $c(i)$ by W. We construct an instance of the standard knapsack problem with $2n$ items having profits $\tilde{p}_1, \ldots, \tilde{p}_{2n}$ and weights $\tilde{w}_1, \ldots, \tilde{w}_{2n}$. The profits are defined as

$$\tilde{p}_j := \begin{cases} p_j + P & \text{for } j = 1, \ldots, n, \\ (3n + 1 - j)P & \text{for } j = n+1, \ldots, 2n, \end{cases} \tag{13.12}$$

while the corresponding weights of the new items are defined by

$$\tilde{w}_j := \begin{cases} w_j + W & \text{for } j = 1, \ldots, n, \\ (4n - j)W - c(j-n), & \text{for } j = n+1, \ldots, 2n. \end{cases} \tag{13.13}$$

Finally, the size $\tilde{c}$ of the standard knapsack is $3nW$. The correctness of this transformation was shown in [376]. It is based on the fact that any feasible packing of the (KP) instance can contain at most one "large" item with index j in $n+1, \ldots, 2n$, and hence the index of this large item "encodes" the number of small items. Furthermore, if the (KP) instance has a feasible solution with value at least $(2n+1)P+1$ it must contain exactly one large item with index j in $n+1, \ldots, 2n$, and at least $j-n$ "small" items. It should be noted that the transformation does not exploit the monotonicity of $c(i)$ but is valid not only for (CKP) but for arbitrary capacity functions and hence also for the expanding knapsack problem.

Theorem 13.3.1 *An instance of (KP) with capacity function $c(\sum_{j=1}^{n} x_j)$ has a feasible solution with objective function value z^* if and only if the instance of (KP) defined by (13.12) and (13.13) has a feasible solution with objective function value $z^* + (2n+1)P$.* □

An obvious drawback of this approach is the appearance of very large coefficients in the (KP) instance and the fact that its profits and weights are highly correlated. The negative effect of this correlation on the performance of algorithms for (KP) is discussed in Section 5.4.2 and Section 5.5.

An improvement of this reduction with smaller coefficients was given by Iida and Uno [244]. Based on a more detailed analysis of the conditions required for the correctness of the above reduction they construct a more complicated instance of an equivalent standard knapsack problem. It should be noted that the following coefficients are in some sense minimal with respect to these conditions. As before, the reduction does not depend on the monotonicity of $c(i)$.

Let z' be the solution value of any feasible solution of (CKP), the larger the better, and set $P' := P - p_{\min} - z' + 1$. Assuming the weights to be sorted in increasing order, set

$$W' := \max\left\{ \max_{j=1,\ldots,n-1} \{c(j) - \sum_{\ell=1}^{j+1} w_\ell\} + 1, 0 \right\},$$

and let $c' := \max_{i \neq j}\{c(i) + c(j)\} + 1$. Clearly, for (CKP) there will always be $c' = c(1) + c(2) + 1$. Then the profits $\tilde{p}_j$ and weights $\tilde{w}_j$ of an instance of (KP) with $2n$ items are defined as follows:

$$\tilde{p}_j := \begin{cases} p_j + P' & \text{for } j = 1,\ldots,n, \\ (2n+1-j)P' + p_{\min} & \text{for } j = n+1,\ldots,2n, \end{cases} \tag{13.14}$$

$$\tilde{w}_j := \begin{cases} w_j + W' & \text{for } j = 1,\ldots,n, \\ (3n-1-j)W' - c(j-n) + c', & \text{for } j = n+1,\ldots,2n. \end{cases} \tag{13.15}$$

The capacity $\tilde{c}$ of the standard knapsack instance is $(2n-1)W' + c'$. The correctness of this construction and the conditions it follows from were established by Iida and Uno [244].

A simple dynamic programming approach for (CKP) was given by Pferschy, Pisinger and Woeginger [376]. It is based on the enumeration of solutions with different cardinality and is thus closely related to the dynamic programming scheme for (kKP) in Section 9.7. We will stick again to dynamic programming by profits which can be turned into an *FPTAS* as in Section 9.7.4. It is straightforward to exchange the role of profits and weights and derive dynamic programming by weights.

Let U be an upper bound on the optimal solution value of (CKP). Define the two-dimensional dynamic programming function $y_j(p,\ell)$ for $j = 1,\ldots,n$, $\ell = 1,\ldots,n$ and $p = 0,1,\ldots,U$, as the optimal solution value of the following problem:

$$y_j(p,\ell) := \min\left\{ \sum_{i=1}^{j} w_i x_i \;\middle|\; \sum_{i=1}^{j} p_i x_i = p,\ \sum_{i=1}^{j} x_i = \ell, x_i \in \{0,1\} \right\}. \tag{13.16}$$

As usual an entry $y_j(p,\ell) = w$ means that considering only the items $1,\ldots,j$ there exists a subset of exactly ℓ of these items with total profit p and minimal weight w among all such subsets. Performing the obvious initialization the optimal solution value of (CKP) is given by

$$\max\{p \mid \exists \ell \in \{0,1,\ldots,n\} \text{ with } y_n(p,\ell) \leq c(\ell),\ p = 0,1,\ldots,U\}.$$

The entries of y_{j+1} can be computed from those of y_j for $j = 0,1,\ldots,n-1$ by the obvious analogon of recursion (9.55)

$$y_{j+1}(p,\ell) := \begin{cases} y_j(p,\ell) & \text{if } p < p_{j+1}, \\ \min\{y_j(p,\ell), y_j(p-p_{j+1},\ell-1) + w_{j+1}\} & \text{if } p \geq p_{j+1}. \end{cases} \tag{13.17}$$

Applying the techniques of Section 3.3 the running time complexity of this approach is $O(n^2U)$ with $O(nU)$ space. The related dynamic programming by weights requires $O(n^2 c_{\max})$ time and $O(n c_{\max})$ space with $c_{\max} := \max_{i=1}^{n} c(i)$. More details about dominance relations and state reduction can be found in [376].

Note that the transformation to a standard knapsack problem does not carry over to an *FPTAS* for (CKP) since any relative error ε in the (KP) instance is dominated by the $\varepsilon(2n+1)P$ term which does not depend on the optimal solution value of (CKP). However, it is straightforward to derive an *FPTAS* from (13.16) by a scaling method analogous to Section 9.7.4. The necessary approximate solution for (CKP) can be found e.g. by solving the approximation algorithm LP-approx for (kKP) iteratively for $k = 1,\ldots,n$ and the corresponding capacity $c(k)$. Taking the maximum over all n solution values will clearly yield a $1/2$-approximation.

A branch-and-bound algorithm for (CKP) was given by Posner and Guignard [397]. It was considerably improved by a more recent algorithm due to Fayard and Plateau [139]. The latter applies upper and lower bounds on the number of items in a feasible and in the optimal solution. Furthermore it applies inner and outer linearization of the capacity function $c(i)$ based on the convex hull of the point set $(i,c(i))$, thus generating standard knapsack problems which yield reasonable heuristic solutions and upper bounds on the optimal solution value.

A computational comparison of the approach to transform (CKP) into an equivalent knapsack problem based on (13.12) and (13.13), the direct dynamic programming scheme given in (13.17) and the branch-and-bound algorithm by Fayard and Plateau was performed in [376]. The high correlation between profits and weights in the equivalent (KP) instance causes the failure of standard knapsack algorithms for the transformation approach even for very small instances. Algorithm Minknap (see Section 5.4.2) is well suited for these "hard" (KP) instances and outperforms the branch-and-bound algorithm from [139] in most cases. However, for large problems of (CKP) the coefficients in the equivalent (KP) instance become unpleasantly huge and prevent the execution of Minknap. Both of the other approaches were clearly dominated by the dynamic programming scheme if reductions and bounds are applied.

13.3.8 The Parametric Knapsack Problem

Parametrizations of integer programs have been the subject of active research for many years. An annotated bibliography covering many aspects of this area was collected by Greenberg [195], references to earlier work can be found e.g. in Bank and Mandel [25] and in Nauss [356].

The original motivation for introducing a parameter is the changing of the problem's data as time goes by (cf. [131]). The most common approach is the substitution of the cost coefficients by (linear) cost functions depending on a (time) parameter t. The corresponding optimal solution value function assigns to every parameter value t' the optimal solution value of the corresponding problem with fixed costs given by evaluating all cost functions at t'.

Parametrization of the right-hand side in (1.2), i.e. the replacement of the knapsack capacity c by an arbitrary function $c(t)$, preserves the structure of the standard knapsack problem. It can be solved by running standard dynamic programming by weights with capacity $\max_t c(t)$ and reporting for every t' the solution value $z_n(c(t'))$ of the dynamic programming function (2.7). Note that a different kind of variable capacity, where $c(i)$ is a function depending on the number of packed items, is treated in Section 13.3.7.

If the item *profits* are parametrized by linear functions we get the following *parametric profit knapsack problem*

$$\begin{aligned} \text{(PPKP)} \qquad & \text{maximize} \quad \sum_{j=1}^{n}(a_j t + b_j)x_j \\ & \text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c, \\ & \qquad\qquad x_j \in \{0,1\}, \quad j = 1,\ldots,n, \end{aligned}$$

where a_j and b_j are the real coefficients of the linear profit function of item j such that $a_j t + b_j \ge 0$ for $t \ge 0$, i.e. $a_j \ge 0, b_j \ge 0$. It is well known that the optimal profit value function of (PPKP) given by

$$v(t) = \max\left\{ \sum_{j=1}^{n}(a_j t + b_j)x_j \;\middle|\; \sum_{j=1}^{n} w_j x_j \le c,\ x_j \in \{0,1\},\ j = 1,\ldots,n \right\}$$

is piecewise linear and convex. This property can be visualized easily. The objective function value of every feasible solution is a linear function of t. Obviously, $v(t)$ is the maximum over all these functions. Clearly, the maximum of a set of linear functions is piecewise linear and convex.

If $v(t)$ consists of a linear piece for an interval $t \in [t',t'']$ then the packing of the knapsack remains unchanged for all parameter values in $[t',t'']$. A *breakpoint*, i.e. a value of t where the slope of $v(t)$ changes, also indicates a change of items in the knapsack packing.

For general parametric programs it is known that the number of breakpoints can be exponential in the number of variables. One may hope that (PPKP), i.e. an integer program with only one constraint, shows a more favourable behaviour. But this is not the case as shown by Carstensen [73].

Theorem 13.3.2 *For any k, there exists an instance of (PPKP) with $\frac{1}{2}(9k^2 - 7k)$ variables such that the corresponding profit value function $v(t)$ has $2^k - 1$ breakpoints in the interval $(-2^k, 2^k)$.* □

Parametrizing the item *weights* by linear functions defines the *parametric weight knapsack problem*

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} (v_j t + w_j) x_j \leq c, \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned}$$

As before v_j and w_j are the real coefficients of the linear weight function of item j such that $v_j t + w_j \geq 0$ for $t \geq 0$ implying $v_j \geq 0$, $w_j \geq 0$. In this case, the optimal value function is a monotone decreasing staircase function consisting of intervals, where the function is constant. An example showing that also in this problem formulation the number of intervals may be exponential can be found in [58].

Theorem 13.3.3 *For any number of items n, there exists a parametric weight knapsack problem with all coefficients in the interval* $]0,2^n[$ *which has* 2^n *different values in the optimal value function for* $t \in [1,2^n]$. □

A more tractable problem in the context of parametric knapsacks is the *inverse-parametric knapsack problem* which was discussed by Burkard and Pferschy [58]. Based on the parametric profit knapsack problem we are given a *target value* v^* and ask for the smallest/largest value of the parameter t (or the earliest/latest point in time) such that the optimal value of the resulting knapsack problem is equal to v^* (if such a t exists):

$$\min / \max \{t \mid v(t) = v^*\}.$$

For this inverse problem pseudopolynomial algorithms were developed in [58] based on the general scheme by Megiddo [341, 342]. Furthermore, in this paper various search methods based on the specific properties of the value function $v(t)$ were presented which turn out to be highly successful in computational experiments.

13.3.9 The Fractional Knapsack Problem

There are a number of practical applications where the selection of an item does not only generate a profit but induces also a certain cost not considered in the weight constraint. In particular, any decision problem involving investments will often aim at maximizing the overall rentability instead of the absolute total profit. This means that every item (project) generates a profit p_j and also a cost $c_j \in \mathbb{N}$ in addition to the weight (resource demand) w_j still limited by a capacity c. The objective is to maximize the *ratio* of the total profit over the total costs. Note that for technical reasons we introduce also some positive fixed costs c_0. Formally, the resulting *fractional knapsack problem* is defined as

$$\text{(FKP)} \qquad \text{maximize} \quad \frac{\sum_{j=1}^{n} p_j x_j}{c_0 + \sum_{j=1}^{n} c_j x_j} \tag{13.18}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c,$$
$$x_j \in \{0,1\}, \quad j = 1,\ldots,n.$$

These fractional objective functions are investigated for many types of combinatorial optimization problems. (Note that the same name *fractional knapsack* is sometimes used for the LP-relaxation of (KP).) An extensive survey of this area is given by Radzik [400].

Fractional problems are closely related to parametric problems in the following way. Let us define the following instance of (PPKP) (see Section 13.3.8) with negative ascent values for an instance of (FKP).

$$\text{maximize} \quad \sum_{j=1}^{n} (p_j - c_j t) x_j - c_0 t$$
$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c,$$
$$x_j \in \{0,1\}, \quad j = 1,\ldots,n,$$

As in Section 13.3.8 let $v(t)$ denote the optimal profit value function of this problem for $t > 0$. It is not difficult to check the following relationship between $v(t)$ and (FKP).

Theorem 13.3.4 *z^* is the optimal solution value of (FKP) if and only if $v(z^*) = 0$. The optimal solution vector x^* yielding $v(z^*) = 0$ is also an optimal solution of (FKP).* □

This statement holds for a very general class of optimization problems and seems to go back to Dinkelbach [107]. It is an immediate consequence of Theorem 13.3.4 that (FKP) can be solved by a search procedure on $v(t)$. Beside a trivial binary search procedure the Newton method, which sets $t_{i+1} := v(t_i)$, is the classical approach to this problem. There is a large number of research contributions on the Newton method and in particular about its convergence rate (see e.g. Radzik [399]).

Note that the computation of the root of $v(t)$ is precisely the task of solving an inverse-parametric knapsack problem briefly described in Section 13.3.8 with $v^* = 0$. Recall that $v(t)$ is piecewise linear and convex. These properties can be exploited efficiently in any search procedure as analyzed for the knapsack problem by Burkard and Pferschy [58].

A different approach to find z^* and x^* is the nice method by Megiddo [341]. Without going into the details, this approach can be applied for any fractional problem where the related parametric problem can be evaluated for a fixed parameter t by an algorithm requiring only $add(n)$ additions and $com(n)$ comparisons. Then the fractional problem can be solved in $O(com(n)(add(n) + com(n)))$ time.

Based on dynamic programming a pseudopolynomial algorithm for (FKP) can be derived along these lines from Lemma 2.3.1 requiring $O(n^2c^2)$ time. This complexity can be improved following the ideas of Megiddo [342] as described in [58].

Theorem 13.3.5 *(FKP) can be solved in* $O\left(\frac{n^2c\log c}{\log n+\log\log c}\right)$ *time.* □

For the approximation of fractional combinatorial optimization problems Hashizume et al. [215] present a general method to apply approximation schemes of the related problem with a linear objective function to the fractional case and deal with (FKP) as an example of their approach. This method was improved for the case of (FKP) by Billionnet [36] in two ways. On one hand a better *FPTAS* for the related linear knapsack problem is applied. On the other hand, a $\frac{1}{2}$-approximation algorithm for (FKP) running in $O(n^2)$ time is derived from the LP-relaxation of (FKP). Combining these two features, an *FPTAS* for (FKP) with an overall running time of $O(n^3/\varepsilon)$ is stated.

Moreover, any *FPTAS* for the standard knapsack problem (KP) with running time $f(n,1/\varepsilon)$ can be used to reach an *FPTAS* for (FKP) with running time

$$O\left(n^2+\sum_{k=1}^{\lceil\log(1/\varepsilon)\rceil} f(n,2^{k+2})\right). \tag{13.19}$$

Thus, plugging in the currently best *FPTAS* by Kellerer and Pferschy [267, 266] described in Section 6.2 would yield a running time of

$$O(n^2+1/\varepsilon^2\log(1/\varepsilon)\cdot\min\{n,1/\varepsilon\log(1/\varepsilon)\})$$

for (FKP).

13.3.10 The Set-Union Knapsack Problem

The *set-union knapsack problem* (SUKP) is a generalization of (KP) with respect to the structure of the items: given a set of n items $N := \{1,\ldots,n\}$ and a set of m so-called *elements* $P := \{1,\ldots,m\}$, each item j corresponds to a subset P_j of the element set P. The items j have nonnegative profits p_j, $j = 1,\ldots,n$, and the elements i have nonnegative weights w_i, $i = 1,\ldots,m$. The total weight of a set of items is given by the total weight of the elements of the union of the corresponding element sets. The objective is to find a subset of the items with total weight not exceeding the knapsack capacity and maximal profit. For $Q \subseteq N$ set $P_Q := \cup_{i\in Q}P_i$. Then, (SUKP) can be formally defined as follows:

$$\text{(SUKP)} \qquad \text{maximize} \sum_{j \in Q} p_j$$
$$\text{subject to} \sum_{i \in P_Q} w_i \leq c.$$

(SUKP) was introduced by Goldschmidt, Nehme and Yu [187]. They gave two industrial applications for the problem: one in flexible manufacturing systems and one for the allocation of memory space in data bases.

By reduction from CLIQUE (see Section 13.2) Goldschmidt, Nehme and Yu show that (SUKP) is strongly $\mathcal{NP}$-hard even when $|P_j| = 2$, $p_j = 1$, $w_i = 1$ for $j = 1,\ldots,n$, $i = 1,\ldots,m$ and the so-called *item-edge graph* is bipartite, where the item-edge graph has vertex set $V = \{1,\ldots,m\}$ and edge set $E = \{e_1,\ldots,e_n\}$ with $e_j = P_j$. They also present an exact dynamic programming algorithm with non-polynomial running time and characterize some special instances which admit polynomial algorithms.

13.3.11 The Multiperiod Knapsack Problem

An interesting special case of (d-KP) arising from a budget planning scenario is the so-called *multiperiod knapsack problem* (MPKP) introduced by Faaland [136]. In this generalization of (KP) the n items with profits p_j and weights w_j are partitioned into d mutually disjoint subsets $N_1,\ldots,N_d$. The selection of items is subject to a time horizon where initially at time 0 only the items from N_1 are available with a resource bound c_1. In the next time step additional items from N_2 become available and the total resources are increased to c_2. In general, at any point in time $k \in \{1,\ldots,d\}$ the items from $N_1 \cup \ldots \cup N_k$ may be selected up to a total capacity c_k. The resulting problem can be formulated as an instance of (d-KP) with a lower triangular constraint matrix. The corresponding weights of an item $j \in N_k$ are given by

$$w_{ij} := \begin{cases} 0 & \text{if } i < k, \\ w_j & \text{if } i \geq k. \end{cases} \tag{13.20}$$

It can be assumed w.l.o.g. that $c_1 < \ldots < c_d$.

The solution of the LP-relaxation of (MPKP) can be easily derived by an explicit formula. This was stated by Dudzinski and Walukiewicz [116] for the binary case and by Faaland [136] for the unbounded version of the problem.

Rather straightforward branch-and-bound algorithms were given for (MPKP). Exploiting the simple structure of the LP-relaxation, reductions and the ingredients of a branch-and-bound scheme for the binary case were given in [116], whereas Faaland [136] described and tested an algorithm for the unbounded case. An *FPTAS* based on dynamic programming and scaling is given by Kellerer [262].

14. Stochastic Aspects of Knapsack Problems

The existing literature and the scientific community seem to agree that the main approach to analyze the running time of algorithms and the performance of approximation methods is worst-case analysis. As indicated in Section 1.5 also sound results on the average behaviour of algorithms would be highly appreciated. However, any venture in this direction is bound to run into two major obstacles.

On one hand, it is not at all clear what a practitioner actually means when speaking of average behaviour. Modeling probabilistic frameworks which give a reasonable image of real-world instances is by no means an easy task. Clearly, any such framework will apply only to a particular family of instances and so will the derived results.

On the other hand, even the most straightforward probabilistic models soon reach a high level of mathematical difficulty and thus the range of meaningful results is rather limited. Moreover, many of these results concern only the limit process of n going to infinity and may not be very insightful for problems with a number of items in a medium range.

In this chapter we will give a review of the most interesting results of this area putting emphasis on topics of algorithmic significance. The presentation of technical lemmas and proofs are beyond the scope of this book because of the considerable amount of required knowledge in probability theory and the mathematical skills involved. Instead we restrict ourselves to the illustrative description of the main results and the algorithms they were derived on. We would like to apologize to the mathematical purist for simplifying some results and ignoring some of the subtleties involved in the handling of probabilistic models. Anybody who is interested to get the full details of this area or who wants to do further research in this field is strongly advised to go back to the original papers in order to be sure of taking all the nontrivial technicalities into account.

Throughout this chapter we will assume a familiarity with the basic notions and notations of probability theory as they can be found in classical textbooks such as Feller [141], Chung [84] or Ross [412]. For sake of completeness let us mention that the expected value of a random variable X is denoted as $E[X]$, and $E[X|Y]$ denotes the expected value of X given Y. Moreover, $X \sim \mathcal{U}[0,1]$ means that the random variable X is distributed uniformly in the real numbers between 0 and 1. As introduction and further guidance in the area of probabilistic analysis of algorithms

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

we recommend the books by Hofri [235], Coffman and Lueker [88] and Habib et al. [203].

In general, results in the area of probabilistic analysis can be put into three groups.

1. *Structural results* which give a probabilistic statement e.g. on the optimal solution value, on the number of items in a feasible solution, etc.
2. *Expected performance* of deterministic algorithms with an acceptable worst-case running time which produce e.g. an optimal solution with a certain probability.
3. *Expected running time* of algorithms which always produce solutions of a certain quality but may have an inferior worst-case running time.

According to this classification we will proceed to review results for the knapsack problem. When some or all of the input data is chosen from a probability distribution we will frequently speak of a *stochastic knapsack problem*. Separate sections will be devoted to the treatment of stochastic aspects of the subset sum problem (SSP) (Section 14.6) and of the multidimensional knapsack problem (d-KP) (Section 14.7). At the end of this chapter the *on-line knapsack problem* is introduced and discussed in Section 14.8. Since the basic definition of an on-line problem does not involve probabilities, the on-line knapsack problem could be put into Chapter 13 where other variants of the knapsack problem are considered. However, basically all meaningful results for this problem are of a stochastic nature which makes the inclusion of that problem in the current chapter much more meaningful.

It should be noted that most of the probabilistic models in this chapter assume that the profits and weights values are real numbers instead of integers as in all other chapters of this book. Many of the results in this chapter only hold asymptotically and are denoted by O (upper bound), and Θ (tight value). For a precise definition of these notations we refer to Section 1.5 or any textbook on algorithms, e.g. Cormen et al. [92].

14.1 The Probabilistic Model

The most obvious probabilistic model which leaves the best chances for deriving theoretical results and also corresponds to the most natural idea of a "random" item is the selection of profits and weights from a continuous uniform distribution. Hence, we assume that $p_j \sim \mathcal{U}[0,1]$ and $w_j \sim \mathcal{U}[0,1]$ for all items $j = 1,\ldots,n$. In this way, every item can be seen as a random point in a two-dimensional profit/weight unit square S_U. This model seems to go back to the seminal work by Lueker [310] who suggested the following setup.

To simplify the mathematical analysis the number n of items resp. points of S_U which are generated in this way is not completely fixed in advance. Instead, it is

useful to draw n from a Poisson distribution with parameter $\mathcal{N}$, where $\mathcal{N}$ is a given value for the expected number of items. This means that the items are in fact generated by a Poisson process where the probability that a given region $R \subset S_U$ contains exactly m points is given by the Poisson distribution with $\lambda = \mathcal{N} \cdot \text{area}(\mathcal{R})$ and

$$P(R \text{ contains } m \text{ points }) = \frac{\lambda^m \exp(-\lambda)}{m!}.$$

Moreover, the distributions of points in disjoint regions of S_U are independent. Note that for very large $\mathcal{N}$, it can be expected that n is not too far away from $\mathcal{N}$. Hence, most authors do not distinguish between n and $\mathcal{N}$ since almost all results only hold asymptotically anyway. Therefore, we will not use $\mathcal{N}$ in the following sections but stick to the mainstream literature and use n as the number of generated items.

The capacity c remains to be selected. To reach the effect that in average a constant fraction of the number of items is included in a solution which fills most of the knapsack capacity we set $c := \gamma n$ with $\gamma \in]0, 1[$. The following lemma was shown in [310].

Lemma 14.1.1 *For $\gamma > 1/2$ the following equation holds.*

$$\lim_{n \to \infty} P\left(\sum_{j=1}^{n} w_j \leq \gamma n\right) = 1$$

This basically means that for $\gamma > 1/2$ with probability tending to 1 *all* items can be packed into the knapsack. Informally speaking, for large instances the expected sum of weights of all items is close to $n/2$ and thus very likely to be smaller than the capacity for $\gamma > 1/2$. To avoid these trivial instances we will only consider the case $\gamma \in]0, 1/2[$ from now on. It should be mentioned that many proofs distinguish the cases $\gamma \in]0, 1/6]$ and $\gamma \in]1/6, 1/2[$. This is due to the fact that in the former case the efficiency of the solution of the LP-relaxation x^{LP}, given by

$$e^{LP} = \frac{z^{LP}}{\sum_{j=1}^{n} w_j x_j^{LP}}, \tag{14.1}$$

is greater than 1 with high probability whereas in the latter case it can be expected to be less than 1.

Also variants of this model and completely different stochastic setups will appear throughout this chapter, in particular in Section 14.3.1.

14.2 Structural Results

The efficiency of a branch-and-bound algorithm for any integer linear program usually depends crucially on the difference between the (unknown) integer optimal solution z^* and an upper bound, which is frequently derived by the LP-relaxation z^{LP}.

The larger the resulting *integrality gap* $z^{LP} - z^*$, the larger the number of nodes in the branch-and-bound tree which have to be evaluated by the algorithm. Therefore, it is interesting to investigate this integrality gap also for the knapsack problem.

From a worst-case perspective a simple example was given in Section 2.2 to show that z^{LP} can be almost twice the value of z^*. Hence, the absolute integrality gap for (KP) can be arbitrarily large independently from n.

A different situation arises for the *expected value* of the integrality gap. The first result on this topic was given by Lueker [310]. A not too difficult result in his paper indicates that the ratio $E[z^{LP}]/E[z^*]$ converges to 1 for n tending to infinity under the probabilistic model of Section 14.1. A more involved question concerns the absolute difference for which he showed the following upper bound.

Theorem 14.2.1 *For an instance of (KP) generated by the model from Section 14.1 with an expected number of n items the expected integrality gap*

$$E[z^{LP} - z^*] \text{ is in } O(\log^2 n/n).$$

In fact, Lueker did not consider z^* explicitly but gave an approximation algorithm producing a lower bound $z^A \leq z^*$ and then bounded $E[z^{LP} - z^A]$. The approximation algorithm considers the items sorted in decreasing order of efficiencies according to (2.2) and packs the items into the knapsack (as in Greedy) but stops as soon as the remaining knapsack capacity falls below $2/3 \log_4 n$. Then it switches to a partial enumeration procedure where disjoint sets of $2 \log_4 n$ successive items are iteratively considered. For each such set all possible subsets are enumerated. The whole procedure is continued until a subset is detected which fills the remaining knapsack capacity "almost" completely. To prove Theorem 14.2.1 it is shown that the probability of failing to find a set almost filling the knapsack capacity is very small and that the expected decrease of efficiencies of the items which are finally packed into the knapsack in the enumeration phase is $O(\log n/n)$. Multiplying this decrease of efficiency of the items packed during this phase with the capacity of roughly $\log n$ which they fill, yields the result.

Lueker [310] also gave a lower bound proportional to $1/n$ for the expected integrality gap if $\gamma \in]1/6, 1/2[$. This result was improved by Goldberg and Marchetti-Spaccamela [186] to a lower bound matching the upper bound, which can be combined into the following statement.

Theorem 14.2.2 *For an instance of (KP) generated by the model from Section 14.1 with an expected number of n items the expected integrality gap*

$$E[z^{LP} - z^*] \text{ is in } \Theta(\log^2 n/n).$$

This means that the integrality gap should decrease with increasing problem size. Such a behaviour was observed also in empirical studies such as Balas and Zemel

[22] and can be intuitively explained by the fact that a large set of items is likely to contain a large pool of items with an efficiency close to the split item and thus increases the chances to fill the capacity with such items. Note that the estimation of the integrality gap in Theorem 14.2.2 will be exploited algorithmically in Section 14.3. The related topic of characterizing the gap between z^* and the solution value z^G computed by the Greedy heuristic will be discussed in Section 14.4.

A different kind of structural statement is related to the core concept which was discussed in detail in Section 5.4. This approach was based on the fact that x^* and x^{LP} were observed to differ only in a small number of items which are located near the split item. Experiments on the actual core size of (KP) under various distributions of the input data can be found in Table 5.1. It was even conjectured in [22] that the expected core size is constant. However, this is not the case since the core size is naturally at least as large as the number of items which have to be changed (i.e. packed or unpacked) when moving from x^{LP} to x^*. And this number of changes denoted by *ch* was shown by Goldberg and Marchetti-Spaccamela [186] to grow logarithmically in expectation with increasing problem size. In particular, they showed the following theorem for the probabilistic model of Section 14.1.

Theorem 14.2.3 *Let $f(n)$ be a monotonous increasing unbounded function and α any constant with $\alpha > 2$. Then*

$$\lim_{n\to\infty} (P(ch \leq f(n)\log n)) = 1,$$

$$\lim_{n\to\infty} \left(P\left(ch \geq \frac{\log n}{\alpha \log\log n} \right) \right) = 1.$$

Not only the integrality gap but also the expected absolute value of z^{LP} resp. z^* itself was investigated by Lueker [311]. Since the knapsack capacity in the probabilistic model of Section 14.1 grows proportional with n, it can be expected that also $E[z^{LP}]$ and $E[z^*]$ grow linearly in n. Lueker managed to derive very precise expressions for the growth rate of $E[z^{LP}]$ and $E[z^*]$ such that under certain fairly general conditions the explicit values for these expectations can be derived for any given instance of (KP) under various distributions.

For a the same model but with $c = 1$ it was shown by Frieze and Clarke [156] that z^* is asymptotically equal to $\sqrt{2n/3}$ with probability going to 1 as n tends to infinity.

A characterization of the asymptotic behaviour of z^* under a very general probabilistic model where the profits and weights are drawn independently as pairs from the same common distribution was investigated by Mamer and Schilling [316]. Evaluating their general result for the above model yields exactly the same result of $\sqrt{2n/3}$ as derived by Frieze and Clarke [156].

14.3 Algorithms with Expected Performance Guarantee

A polynomial "almost always optimal" algorithm was presented by Goldberg and Marchetti-Spaccamela [186]. Their algorithm GMS is quite unusual in the field of expected performance analysis because it does not only produce a solution which is "good" or optimal with high probability but computes - again with high probability - a solution which is *certified* to be optimal. Hence, it is really known at the end of the execution whether the algorithm managed to find the optimal solution or returns an approximate solution.

In our description of GMS in Figure 14.1 there are two parameters β and $m(n,k)$ which follow from existence results in [186] and cannot easily be stated explicitly. Unfortunately, there exists no full journal version of this interesting proceedings paper.

Algorithm GMS:
compute z^{LP}, x^{LP}, e^{LP} and split item s
$I := \{j \mid x_j^{LP} = 1\}$
$p_d := \beta(k+2)\frac{\log^2 n}{n}$ *distance measure*
$z^A := 0$
enumerate at most $m(n,k)$ subsets S of $\{1,\ldots,n\} \setminus \{s\}$ with
$$\sum_{j \in S}(p_j - e^{LP} w_j) \leq p_d$$
$X := (S \cup I) \setminus (S \cap I)$
if $\sum_{j \in X} w_j \leq c$ then $z^A := \max\{z^A, \sum_{j \in X} p_j\}$
if less than $m(n,k)$ subsets were found and $z^{LP} - z^A < p_d$ then
z^A is the optimal solution value
otherwise
z^A is an approximate solution value

Fig. 14.1. Polynomial algorithm GMS by Goldberg and Marchetti-Spaccamela which computes an optimal solution with high probability.

The main idea of algorithm GMS is quite easy. It is based on the core concept of Section 5.4 and tries to exchange items in and out of the solution of the LP-relaxation but using only those subsets S in these exchanges which have a total profit which differs not too much from a virtual item of the same weight $w(S)$ and with the same efficiency as the split item s.

Under the probabilistic model the number of such subsets will be polynomially bounded with high probability. In particular, the following statement holds.

Theorem 14.3.1 *For every $k > 0$ there are constants β and n_0 and a polynomial $m(n,k)$ such that for every $n > n_0$* GMS *terminates with a verified optimal solution*

with probability at least $1 - 1/2^k$ *after enumerating at most* $m(n,k)$ *subsets of items.*

This result is derived by considering a slightly modified version of GMS to facilitate the probabilistic analysis. The modification consists basically of replacing the efficiency value e^{LP} (defined in (14.1)), which is crucial for the selection of subsets S, by an approximation.

A drawback of Theorem 14.3.1 is the fact that the involved polynomial $m(n,k)$ will have an extremely large exponent which prohibits any practical application of this algorithm (see the comment in Beier and Vöcking [30]). Clearly, one can also change GMS into an exact algorithm by dropping the polynomial cardinality restriction on the number of subsets. In this case, we would get an exact algorithm which has a polynomial running time with probability arbitrarily close to 1. However, the amount of deviation of this probability from 1 carries over into the required constants and increases the magnitude of $m(n,k)$. It was pointed out in [30] that the enormous exponents of the polynomial do not permit an expected polynomial running time of the resulting exact algorithm. This aim will be reached in Section 14.5.

14.3.1 Related Models and Algorithms

A completely different model of a stochastic knapsack problem was treated by Goel and Indyk [185]. They consider the case where the item weights in a particular instance are not fixed constants but random variables W_j with a given distribution function F_j. In this setup the constraint (1.2) of (KP) turns into a probability condition with a given *overflow probability* π

$$P\left(\sum_{j=1}^{n} W_j x_j \leq c\right) \geq 1 - \pi. \tag{14.2}$$

A deterministic algorithm should decide which items to pack based on the given constant profit values p_j and the distribution functions F_j, respectively the expected values $E[W_j]$, but without knowing the actual realisation of W_j.

For the case where the W_j are drawn from exponential distributions a polynomial time approximation scheme is presented in [185] in the sense that the computed solution value is at least as large as the optimal solution value but violates the weight constraint with probability at most $\pi(1+\varepsilon)$. The algorithm shares the main characteristics with the *FPTAS* for (KP) as introduced in Section 2.6 and refined in Section 6.2.

Along the same lines an approximation algorithm is also given for the case of the Bernoulli distribution, where the weights are either w_j (a constant) or 0. In this case the probability π is assumed to be a constant in the running time analysis.

Note that for the Poisson distribution any deterministic approximation algorithm can be easily applied to this type of stochastic knapsack problem because of the summation property of the Poisson distribution.

The same stochastic model of random weights represented by random variables W_j was considered in the numerical experiments of the more general paper by Kleywegt, Shapiro and Homem-de-Mello [278]. Instead of maximizing the profit under an overflow probability as in (14.2), it is suggested in [278] to take a possible overflow into account in the objective function. To be more precise, any subset of items is a feasible solution but a penalty has to be paid for a violation of the capacity constraint. This penalty is assumed to be linear with a constant factor q for each unit of overflow. Thus, the objective function is given by the following combination of profit values and an expected value:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j - q \cdot E\left[\max\left\{0, \sum_{j=1}^{n} W_j x_j - c\right\}\right]. \tag{14.3}$$

This variant of a stochastic knapsack problem is considered by Kleywegt et al. [278] as an illustrative example for the application of a Monte Carlo simulation-based method for stochastic discrete optimization. In this approach an approximate solution is determined by generating a large number K of realizations w_j^k of the n random variables W_j with $k = 1, \ldots, K$. Then the values of x are computed subject to the following average objective function over all K iterations which should provide a reasonable approximation of the expected value in (14.3).

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j - \frac{q}{K} \sum_{k=1}^{K} \max\left\{0, \sum_{j=1}^{n} w_j^k x_j - c\right\} \tag{14.4}$$

The computational experiments reported in [278] consider normally distributed weights W_j with means μ_j and standard deviations σ_j generated from a uniform distribution. Note that for the normal distribution (14.3) can be easily evaluated for any given x by the summation of normal distributions. This permits the computation of an optimal solution for comparisons (see below). The experiments were extensively analyzed, in particular to characterize the influence of the magnitude of K. Among the theoretical results it was shown that the sample size K required to reach a given accuracy of the solution increases logarithmically in the number of feasible solutions, which would be linear in n. However, in the experiments the parameters of the generated knapsack instances such as correlation of profits and expected weights turned out to be dominating this relation. A different Monte Carlo simulation procedure for the case of random profits and constant weights (see below) was given by Morton and Wood [352].

The basic model of random weights and constant profits was slightly extended by Cohn and Barnhart [89] in the following way. The weights are still random variables

W_j with a given distribution function F_j. However, the profits of an item j are now given by the weight multiplied by a constant scaling factor s_j. Hence, the profits are dependent on the same distribution as the weights. Again, a deterministic algorithm should select a subset of items without knowing the realizations of the n random variables. The objective function considered in [89] is almost similar to (14.3) except that the expectation must be taken also over the profits. Thus, the objective function is given by the following expected value.

$$\text{maximize} \quad E\left[\sum_{j=1}^{n} s_j W_j x_j - q \cdot \max\left\{0, \sum_{j=1}^{n} W_j x_j - c\right\}\right] =$$

$$= \sum_{j=1}^{n} s_j E[W_j] x_j - q \cdot E\left[\max\left\{0, \sum_{j=1}^{n} W_j x_j - c\right\}\right]$$

Cohn and Barnhart [89] deal with the case where the F_j are independent normal distribution functions, but not necessarily identical. Based on dominance between the mean and variance of different W_j and on the construction of upper and lower bounds a preliminary branch-and-bound procedure is introduced.

Another type of stochastic knapsack problem with reversed roles of constants and random values was discussed by a number of authors starting (to the best of our knowledge) with the paper by Steinberg and Parks [445] in 1979 and continued in the work of Sniedovich [438], [439], Carraway, Schmidt and Weatherford [72], Morita, Ishii and Nishida [351], Henig [222], Morton and Wood [352] and others. The general model arises in the context of a financial decision problem where every item corresponds to an investment project with a fixed investment requirement w_j but stochastic future profits P_j which implicitly include the involved risk. Instead of maximizing the total profit different risk criteria are considered such as maximizing the probability that the total profit reaches a given target. Financial optimization problems with stochastic risk respectively profit estimations and deterministic budget constraints are a classical topic in finance. It is beyond the scope of this book to enter into this area for which a number of textbooks are available such as Korn [285] and Dupacova, Hurt and Stepan [118].

14.4 Expected Performance of Greedy-Type Algorithms

A number of authors dealt with the performance of Greedy or close relatives, in particular simplifications, of this basic approach introduced in Section 2.1. The motivation for this research can be found in the discrepancy between the relatively good practical performance of greedy-type methods and the disappointing worst-case behaviour (see Section 2.5). A further natural reason is the fact that the resulting probabilistic analysis is more tractable for these very simple algorithms. An intuitive

geometric construction of the probabilistic setup for Greedy can be found in Frieze and Reed [157, Section 5.3]. This might be a suitable starting point into the field of probabilistic analysis for the less experienced reader.

The most basic method to fill a knapsack is Greedy without sorting. We will refer to this packing of the items in an arbitrary order as Unsorted-Greedy (cf. Section 2.1). Since even Greedy has an arbitrarily bad worst-case performance ratio, Unsorted-Greedy can perform as bad as possible by the same argument. However, if the items are generated independently from a certain distribution, better results can be expected.

Let z_k resp. W_k be the profit resp. weight of the knapsack packing attained after considering the first k items (note that item 1 will be packed in any case).

Szkatula and Libura [451] considered a probabilistic model slightly different from Section 14.1. For the case where the weights are again uniformly distributed, i.e. $w_j \sim \mathcal{U}[0,1]$ for all items $j = 1,\ldots,n$, and the capacity is fixed by $c = 1$ they proved the following result.

Theorem 14.4.1 *Let p_j be independently identically distributed from a common arbitrary distribution with expected value V and expectation of the squared profits given by $\bar{V}$. Then the following expectations and variances are attained by* Unsorted-Greedy.

$$E[W_k] = \tfrac{k}{k+1}, \qquad Var[W_k] = \tfrac{k}{(k+2)(k+1)^2},$$

$$E[z_k] = V\textstyle\sum_{j=1}^{n} \frac{1}{j}, \quad Var[z_k] = \bar{V}\sum_{j=1}^{n} \frac{1}{j} - V^2 \sum_{j=1}^{n} \frac{1}{j^2},$$

$$E[z_k] \approx V \log n, \qquad Var[z_k] \approx \bar{V} \log n - V^2 \tfrac{\pi^2}{6}.$$

The probability of packing item k is given by

$$P(x_k = 1) = \frac{1}{k} \quad \text{for } k = 1,\ldots,n.$$

For the original sorted version Greedy a weaker statement with the same asymptotic behaviour was shown by Szkatula and Libura in [452] (see also the experimental comparisons reported in [451]). If the weights w_j are chosen independently identically from an arbitrary distribution over $]0,1]$ then the total weight of the items packed by Greedy is asymptotically equal to any fixed knapsack capacity c with probability 1 as n goes to infinity. This behaviour should not be surprising since it is very likely that any gap between the weight of a partial solution and the knapsack capacity can be (almost) filled if the set of remaining items is extremely large.

The obvious superiority of Greedy over Unsorted-Greedy can also be illustrated by the following statement from [452]. Under certain technical assumptions, the ratio of the objective function values z^G/z^* is asymptotically equal to 1 with probabil-

ity 1 as n goes to infinity. However, for the solution value z_n computed by Unsorted-Greedy it could be shown under several probabilistic scenarios that the ratio z_n/z^* is asymptotically equal to 0 in the same sense.

A different analysis of the performance of Greedy was given by Diubin and Korbut [108]. Assuming as in Section 14.1 that $p_j \sim \mathcal{U}[0,1]$ and $w_j \sim \mathcal{U}[0,1]$ for all items $j = 1,\ldots,n$, they derived an asymptotic result for the gap between z^G and z^* which depends on the knapsack capacity c.

Theorem 14.4.2 *If $c > (1/2 - t/3)n$ for some fixed $t > 0$ then*

$$\lim_{n\to\infty} \left(P\left(z^* - z^G \leq t\right)\right) = 1.$$

For the given model we always have $z^* - z^G \leq 1$ (see Corollary 2.2.3). Hence, the statement of the theorem is relevant for $t \in]0,1[$, i.e. $c \in]1/6n, 1/2n[$. Note that for $t = 0$ this statement is closely related to Lemma 14.1.1 since all items can be expected to be packed into the knapsack in that case.

A similar result holds also for Greedy-Split where Greedy is stopped after finding the split item s and only items $1,\ldots,s-1$ are packed. An extension of the extremely tedious calculations to general distributions was given in [109].

A comparison of Greedy to the solution value of the LP-relaxation was recently performed by Calvin and Leung [61]. They showed that the absolute difference between the two solution values is of order $\sqrt{n}$ under the probabilistic model of Section 14.1 with $c = \gamma n$. This can be compared (although not in a mathematically rigorous sense) to the result of Theorem 14.2.2 which gives an order of $\log^2 n/n$ estimation for the difference between the solution value of the LP-relaxation and the optimal solution value.

To state the result more precisely, recall from Section 14.1 that the cases $\gamma > 1/6$ and $\gamma \leq 1/6$ often have to be distinguished (as in Theorem 14.4.2). This is also the case in [61] where the authors define a constant K depending on γ with

$$K := \begin{cases} 1/\sqrt{6\gamma} & \text{for } \gamma \in]0, 1/6], \\ 3/2 - 3\gamma & \text{for } \gamma \in]1/6, 1/2[. \end{cases} \tag{14.5}$$

It can be shown that K converges in probability to e_s, i.e. the efficiency of the split item, which means that for any $\delta > 0$ there is

$$\lim_{n\to\infty} (P(|K - e_s| > \delta)) = 0.$$

For the definition of K given in (14.5) the following asymptotic result of convergence in distribution was shown by Calvin and Leung [61].

Theorem 14.4.3

$$\lim_{n\to\infty}\left(P\left(\sqrt{\frac{n}{K}}\left(z^{LP}-z^{G}\right)\leq x\right)\right)=F(x),$$

for a continuous distribution function $F(x)$.

Of course the precise behaviour of $z^{LP}-z^{G}$ would become clear only with the knowledge of $F(x)$, which does not appear explicitly in the proof. From computational experiments it is conjectured in [61] that $F(x)$ is similar to $1-\exp(-x^2/2)$.

14.5 Algorithms with Expected Running Time

Gaining results about the expected running time behaviour of exact algorithms for the knapsack problem has been a point of major interest for many years (see the comment by Lueker in [310]). The good practical performance of many algorithms for uniformly random data suggested that polynomial expected running time might be achievable.

A result in the direction of measuring the expected behaviour of an exact algorithm for the knapsack problem is due to Borgwardt and Brzank [41]. They analyzed under a very special stochastic model the expected running time of an enumeration scheme depending parametrically on the number of undominated states, i.e. pairs of profit and weight values, which are attained by an item set (see Section 3.4).

A major breakthrough in this field is the recent paper by Beier and Vöcking [30]. Pursuing the line of research of Borgwardt and Brzank they also bounded the number of undominated states under more general probabilistic assumptions and combined the resulting structural results with an algorithm similar to DP-with-Lists introduced in Section 3.4. Recall from the Lemma 3.4.3 that (KP) can be solved in $O(nB)$ time, if B is a bound on the number of undominated states.

The probabilistic model introduced by Beier and Vöcking [30] is more general than the previously discussed models. In particular, there are no assumptions on the item weights but only on the profits. For the case of uniformly distributed profits, i.e. $p_j \sim \mathcal{U}[0,1]$, and arbitrary weights (and capacity) they show the following polynomial bound on the number of undominated states.

Theorem 14.5.1 *The expected number of undominated states is in* $O(n^3)$.

This result can be combined immediately with Lemma 3.4.3 to the following central result of Beier and Vöcking [30] for uniformly distributed profits.

Corollary 14.5.2 *(KP) can be solved optimally in* $O(n^4)$ *expected running time.*

An analogous result for exponentially distributed profits yields an algorithm with expected running time $O(n^3)$.

It should be noted that also substantial generalizations of Theorem 14.5.1 were derived. The main result of Beier and Vöcking [30] is the following pseudopolynomial bound.

Theorem 14.5.3 *Let the item profits p_j be drawn independently from continuous distributions F_j with density functions $f_j(x)$ for $x \in \mathbb{R}^+$. Define ϕ such that*

$$\phi \geq \max_{j=1}^{n} \left(\max_{x>0} f_j(x) \right)$$

and $E_{\max} := \max_{j=1}^{n} E[p_j]$. Then for arbitrary weights w_j the expected number of undominated states is bounded by

$$O(\phi E_{\max} n^4).$$

This result gives an interesting intuition about the positive influence of "randomness" on the number of undominated states. After scaling the profits by $E_{\max}$, the bound of Theorem 14.5.3 is given by $O(\phi n^4)$. The maximum density value ϕ can be interpreted as a measure for the randomness of the instance. If ϕ is very large or goes to infinity, there is almost no random influence left and the expected number of undominated states can be exponential (for an appropriate choice of weights). However, if ϕ is bounded from above by a (small) constant, the instances contain "a high degree of randomness" and can be solved in polynomial expected time as in Theorem 14.5.1 with $\phi = 1$.

Beier and Vöcking [30] also extend their result to the case of discrete distributions yielding a bound of the same order. Furthermore, an analogous analysis can be applied to the somehow symmetric case where the weights w_j are drawn from a given distribution and the profits are arbitrary but deterministic.

14.6 Results for the Subset Sum Problem

For the subset sum problem discussed in Chapter 4 a number of probabilistic studies are known. However, we will restrict ourselves to a brief survey of results. The main point of attention in this area is the analysis of simple greedy-type algorithms and in particular the distribution and expected value of the gap g remaining between the knapsack capacity and the solution value attained by an approximation algorithm, i.e. $g = c - z^A$. Clearly, $z^* - z^A \leq g$.

D'Atri and Puech [99] considered the application of Greedy-Split to (SSP), i.e. a greedy algorithm where the items are packed only until the split item is reached.

Recall that the resulting solution is identical to the rounded down solution of the LP-relaxation of (SSP). An extended version called Ext-Greedy-Split adds an additional item to the solution of Greedy-Split namely the largest remaining item which does not exceed the the remaining capacity and hence fits into the knapsack.

In their probabilistic model a fixed upper bound b is given for the item weights which are chosen from the discrete uniform distribution $w_j \sim \mathcal{U}\{1,\ldots,b\}$. The knapsack capacity is chosen discrete uniformly with $c \sim \mathcal{U}\{1,\ldots,nb\}$, which means that all items fit into the knapsack with probability roughly $1/2$ in analogy to Lemma 14.1.1.

The main result of d'Atri and Puech [99] is the exact probability distribution of the gap g remaining between the solution value of Ext-Greedy-Split and the capacity c. As a corollary they derive the following asymptotic probabilities:

Corollary 14.6.1 *For* Ext-Greedy-Split *without sorting there is*

$$P(g > 0) \approx b/(2n),$$

whereas after sorting the items in decreasing order of weights there is

$$P(g > 0) \approx 1/2 \exp(-n/b).$$

This means that for n going to infinity the probability of filling the knapsack completely (which guarantees an optimal solution) tends to 1. Under the same probabilistic model d'Atri and di Rende [98] listed further results. They also continued the analysis of the behaviour of the greedy algorithm (without sorting).

Theorem 14.6.2

$$P(\text{Ext-Greedy-Split } \textit{does not solve (SSP) to optimality}) \leq \frac{1}{2}\left(1-\frac{1}{b}\right)^n$$

This analysis was extended by Pferschy [375] who computed upper and lower bounds for expected values of various relevant parameters of the two greedy variants. Among other results, the following bounds for $E[g]$ under the condition that the problem is non-trivial were shown by Pferschy [375].

Theorem 14.6.3 *For* Greedy-Split *(with and without sorting) there is*

$$E\left[g \,\middle|\, \sum_{j=1}^{n} w_j \geq c\right] = \frac{1}{3}(b-1).$$

The gap g of Ext-Greedy-Split *without sorting is bounded by*

$$\frac{b(\ln b - 6/5)}{2n} \leq E\left[g \,\middle|\, \sum_{j=1}^{n} w_j \geq c\right] \leq \frac{b \ln b}{2n}$$

for larger values of n and b. For Ext-Greedy-Split *in decreasing order of weights there is*

$$\frac{1}{2}\left(1-\frac{1}{b}\right)^n \leq E\left[g \,\middle|\, \sum_{j=1}^{n} w_j \geq c\right] \leq \frac{1}{2}\left(1-\frac{1}{b}\right)^n \left(1+\frac{b}{n}\right).$$

Almost the same scenario was investigated by Tinhofer and Schreck [459]. In their probabilistic model the upper bound b may be a function $b(n)$ as long as it is increasing slower than n, i.e. $\lim_{n\to\infty} b(n)/n = 0$. They investigate the original algorithm Greedy, which continues to pack items after reaching the split item whenever they fit. Beside analyzing the gap g the main result in [459] says that the randomized version of Greedy, where the items are considered in a random order, will fill the knapsack completely, and hence solve (SSP) to optimality, with probability $1/2$ for almost all problem instances generated under the stochastic model.

A comprehensive treatment of greedy-type algorithms for (SSP) from a stochastic point of view is due to Borgwardt and Tremel [42]. Their paper is recommended as starting point for further reading. They argue convincingly for the use of a different probabilistic model with a continuous uniform distribution where $w_j \sim \mathcal{U}[0,1]$ and $c \sim \mathcal{U}[0,n]$. The probability that all items can be packed into the knapsack is exactly $1/2$, almost as in the model of d'Atri and Puech. Therefore, all investigations are performed under the condition that $\sum_{j=1}^{n} w_j \geq c$. The most interesting and astonishing result of [42] seems to be the following fact.

Theorem 14.6.4 *The distribution of g is identical for* Ext-Greedy-Split *and for* Greedy.

Note that this surprising result holds for both probabilistic models introduced in this section.

A full discussion of the complete paper by Borgwardt and Tremel [42] is beyond the scope of this book. Beside the computation of rather complicated distribution functions of g for the greedy variants discussed above, also a modification of Greedy is considered. It will be called Improved-Greedy and can be seen as the following combination of Greedy and Ext-Greedy-Split. At the end of the execution of Greedy the last packed item is removed again and replaced by the largest item which now fits into the knapsack and has an index higher than the previously removed item.

The four greedy variants are analyzed both for the case of arbitrarily ordered items and for the sorted case (in decreasing order of weights). Some of the results are analogous to those in [99], [459] and [375] with the correction of some minor inconsistencies in the first of these papers. The following theorem summarizes the expected values derived for greedy-type algorithms under the continuous probabilistic model used by Borgwardt and Tremel [42].

Theorem 14.6.5
Algorithm Greedy-Split *with and without sorting, cf. Theorem 14.6.3:*

$$E\left[g\left|\sum_{j=1}^{n} w_j > c\right.\right] = \frac{1}{3}$$

Algorithms Ext-Greedy-Split *and* Greedy *without sorting:*

$$E\left[g\left|\sum_{j=1}^{n} w_j > c\right.\right] = \frac{1}{3n} + \frac{1}{4n} + \ldots + \frac{1}{(n+2)n}$$

Algorithm Ext-Greedy-Split *with sorting in decreasing order of weights:*

$$E\left[g\left|\sum_{j=1}^{n} w_j > c\right.\right] = \frac{1}{n+2}$$

Algorithm Improved-Greedy *without sorting for large n:*

$$E\left[g\left|\sum_{j=1}^{n} w_j > c\right.\right] \approx \frac{3}{4}\frac{\ln n}{n}$$

14.7 Results for the Multidimensional Knapsack Problem

The extension of (KP) to the multidimensional problem (d-KP) was the subject of Chapter 9. A number of authors have investigated probabilistic aspects of this special case of a general integer program. The resulting mathematical efforts are extensive and require a certain knowledge of probability theory and stochastic processes. Since (d-KP) is not the central topic of this book we will deal with the results in this area even more cursory than in the previous sections.

The most widely applied probabilistic model chooses all coefficients p_j and w_{ij} from a continuous uniform distribution $\mathcal{U}[0,1]$. The choice of the knapsack capacities c_i differs between authors.

Dyer and Frieze [123] extended the results of Lueker [310] and Goldberg and Marchetti-Spaccamela [186] for (KP) presented in Section 14.2 to the d-dimensional case. It should be noted that this generalization is by no means straightforward but requires different approaches. Following the model for (KP) in their analysis the capacities are fixed to $c_i := \gamma_i n$ with $\gamma_i \in]0, 1/2[$ for $i = 1, \ldots, d$ (cf. Lemma 14.1.1). As generalizations of Theorems 14.2.1 and 14.2.3 the following statements were shown by Dyer and Frieze [123].

Theorem 14.7.1 *There exists a constant k depending only on d and* γ_i *such that*

$$E[z^{LP} - z^*] \leq k \frac{\log^2 n}{n}.$$

As before *ch* denotes the maximum number of changed variables when moving from x^{LP} to x^*. We give an abbreviated formulation of the result by Dyer and Frieze [123].

Theorem 14.7.2 *Let* $g(n)$ *be a monotonous increasing unbounded function proportional to* $\log n$. *Then*

$$\lim_{n\to\infty} (P(ch \geq g(n)\log n)) = 1.$$

Furthermore, an algorithmic construction with a similar flavour as algorithm GMS and a corresponding result analogous to Theorem 14.3.1 was given in [123].

A central point of attention in the stochastic analysis of (d-KP) is the behaviour of the optimal solution value z^*.

Improving upon an earlier analysis by Frieze and Clark [156] the behaviour of z^* was characterized by Schilling [424]. In their probabilistic setup the capacities were fixed to $c_i = 1$. Schilling showed that z^* is with high probability asymptotically equal to $z(n,d)$, where

$$z(n,d) := (d+1)\left(\frac{n}{(d+2)!}\right)^{1/(d+1)}. \tag{14.6}$$

More precisely, the result in [424] (also announced in [316]) is stated as:

Theorem 14.7.3 *There exists a sequence* $\delta(n)$ *with* $\lim_{n\to\infty} \delta(n) = 0$ *such that*

$$\lim_{n\to\infty} P\Big(z(n,d)(1-\delta(n)) \leq z^* \leq z(n,d)(1+\delta(n))\Big) = 1.$$

The nature of this convergence of z^* to $z(n,d)$ was further analyzed and presented in a tighter formulation by Schilling [425].

A generalization of this result was given by Szkatula [449] who replaced the fixed capacities $c_i = 1$ by functions $b_i(n)$. Under certain conditions and bounds on these functions he managed to generalize the above expression of $z(n,d)$ yielding an analogon of Theorem 14.7.3 with

$$z(n,d) := (d+1)\left(\frac{n \cdot b_1(n) \cdot \ldots \cdot b_d(n)}{(d+2)!}\right)^{1/(d+1)}. \tag{14.7}$$

This result was further extended to cover also very large values of $b_i(n)$ in a sequel paper of Szkatula [450].

A more general probabilistic model with p_j resp. w_{ij} independently and identically distributed with two arbitrary distribution functions P for profits and W for weights was considered by Meanti et al. [340]. In analogy to the previous model the capacities were fixed as $c_i = \gamma_i n E[W]$ with $\gamma_i \in]0,1[$ for $i = 1,\ldots,d$. In this model they showed that with probability 1, the optimal solution value z^* is asymptotically equal to $z(LD(\{1,\ldots,d\}))$, i.e. the upper bound derived by the Lagrangian dual (see Section 9.2). Moreover, in this way z^* converges to a function of the d capacities c_i. The rates of this convergence were established in a sequel paper by van de Geer and Stougie [466].

For the same model it was shown by Rinnooy Kan, Stougie and Vercellis [407] that the generalized greedy algorithm described in Section 9.5.1 yields an approximate solution value z^A converging to the optimal solution, i.e.

$$P\left(\lim_{n\to\infty} \frac{z^A}{z^*} = 1\right) = 1.$$

Averbakh [15] used the same probabilistic model with general capacities to study the efficiency of the Lagrangian relaxation for (d-KP) (see again Section 9.2). Among other results he formulated sufficient conditions such that the objective function value of a Lagrangian relaxation is close to the optimal solution value with high probability. More precisely, one of his results says that there exist Lagrangian multipliers λ such that the probability that $z(L(\lambda,\{1,\ldots,d\}))$ is a feasible ε–approximation of z^* tends to 1 as n goes to infinity. In [15] further characterizations and expectations of the Lagrangian dual solution for specific input distributions are derived.

A more exotic probabilistic model where all profits are fixed to $p_j = 1$, the knapsack capacities are fixed to a constant $c_i = b$ and only the weights are drawn from the continuous uniform distribution $w_{ij} \sim \mathcal{U}[0,1]$ was considered by Fontanari [148]. In this paper the so-called *profit density* z^*/n is characterized approximately as a function of b and d. It can be easily seen that for large n there is $\lim_{d\to\infty} z^*/n = 2b/n$.

14.8 The On-Line Knapsack Problem

A completely different point of view of the knapsack problem arises if we assume that the data of the items and even their number is unknown at the beginning of the packing process. In an on-line scenario the items can be seen as arriving one after the other (like on a conveyor belt). As soon as an item arrives and its data becomes known it has to be decided immediately whether to pack this item (and keep it in the knapsack forever) or discard the item (and never get a chance to consider it again). The number of items n may, or may not be known at the beginning.

On-line versions of combinatorial optimization problems are a widely studied field especially in the area of scheduling and bin packing. Introductions and surveys to the topic were given e.g. by Borodin and El-Yaniv [43] and in the volume edited by Fiat and Woeginger [145].

The main focus of theoretical investigations of on-line problems is a worst-case analysis where the output of a given on-line algorithm is compared with the optimal solution derived by an off-line method which knows the complete input data form the beginning. This so-called *competitive analysis* searches for the *competitive ratio* of an on-line algorithm, which is similar to the relative worst-case performance ratio of an approximation algorithm (see Section 2.5).

It could be argued that the on-line variant of (KP) should be treated among the other variations in Chapter 13. However, for the knapsack problem no deterministic analysis of algorithms for the on-line case is known to the authors. Obviously, this is due to the fact that the worst-case behaviour of any strategy for on-line (KP) is bound to be as bad as possible as illustrated by the following simple adversary example (a more sophisticated example can be found in Marchetti-Spaccamela and Vercellis [319]).

All items have weight c, i.e. any algorithm can only choose a single item. The first item to arrive has profit M. If the on-line algorithm decides to pack this item into the knapsack, all the following items will have a very large profit value M^2. None of these items can be added into the knapsack whereas the optimal strategy would pack one of these items and thus reach a performance ratio of $M^2/M = M$. If the on-line algorithm discards the first item, we are back at the beginning and again receive an item with profit M. This scenario can be repeated until either an item of this type is packed and we finish as above by sending very large items with profit M^2, or until all n items are discarded and the on-line algorithm terminates with an empty knapsack.

Frequently, it is assumed that n is known at the beginning of the on-line process. In this case, no algorithm will stop with an empty knapsack but pack at least the last item. However, we can still reach the performance ratio of M by sending at the end (if the on-line algorithm did not pack any of the $n-1$ previous items) an item with profit 1. The optimal algorithm would have packed one of the items with profit M, whereas the on-line algorithm is left with only profit 1.

It might be an interesting open problem to find suitable restrictions on the pure on-line formulation which make sense from a real-world point of view and permit the construction and analysis of more successful algorithms. Models of this type are known as *semi-on-line problems* and have been developed and explored in recent years e.g. for scheduling problems (see e.g. Kellerer et al. [264])

The negative outlook on to the deterministic on-line knapsack problem brightens up to some extent when a probabilistic point of view is taken. Indeed, if the distribution of profits and weights is given, then we have some information at hand to judge

whether an item is "good" or "bad", meaning whether its efficiency is high or low. This gives rise to a simple greedy-type on-line algorithm Threshold with objective function value z^T which packs all items in a greedy way if their efficiency exceeds a given threshold efficiency T. It is illustrated in detail in Figure 14.2.

Algorithm Threshold:

$W := 0$ *total weight of the currently packed items*
$z^T := 0$ *profit of the current solution*
for $j := 1$ to n do
  if $W + w_j \leq c$ and $p_j/w_j \geq T$ then
    $x_j := 1$
    $W := W + w_j$
    $z^T := z^T + p_j$
  else $x_j := 0$

Fig. 14.2. Algorithm Threshold packs items on-line if they fit and if their efficiency exceeds the bound T.

The corresponding average-case behaviour, always assuming the a-priori knowledge of n, was studied extensively in two interesting papers. Assuming the probabilistic model of Section 14.1, Marchetti-Spaccamela and Vercellis [319] proved the following asymptotic bound on the expected behaviour of a relaxed version of Threshold where a fractional part of an item can be used to fill the knapsack completely. For this LP-relaxation of Threshold with solution value $\bar{z}^T$ they choose T as the asymptotic efficiency of the split item as given in Section 14.4, a very natural choice, and show:

Theorem 14.8.1 *Setting $T := K$ from (14.5) the expected gap*

$$E[z^{LP} - \bar{z}^T] \text{ is in } O(\sqrt{n}).$$

It can also be shown that the above choice of T is asymptotically best possible, i.e. $E[z^{LP} - \bar{z}^T]$ is at least of order $\sqrt{n}$ for any T. Note that because of the linear asymptotic growth of $E[z^{LP}]$ and $E[z^*]$ (cf. Section 14.2) Theorem 14.8.1 implies that the relative error of z^T compared to z^* tends to 0 as n goes to infinity because the absolute error grows only with $\sqrt{n}$.

An improvement of Threshold can be reached by an *adaptive threshold algorithm*. In this approach the threshold T is updated during the execution of the algorithm depending on the current filling of the knapsack. In particular, one wants to control the increase of weight of the current knapsack filling during the algorithm. If the currently packed items are below the total weight expected at this point, there is a risk that the knapsack is hardly full at the end of the algorithm and we should thus lower the threshold T to fill up the knapsack. On the either hand, if the current

filling is above the exepectation we can increase T to improve the efficiency of the packing.

For the sake of feasibility of a probabilistic analysis it is suggested to perform such an update only in stages after considering fixed blocks of items to gain stochastic independence. For this variant Marchetti-Spaccamela and Vercellis [319] showed that for suitably chosen parameters the performance of the resulting algorithm improves upon Theorem 14.8.1 considerably yielding

$$E[z^{LP} - \bar{z}^T] \text{ is in } O(\log^{3/2} n). \tag{14.8}$$

They also showed that this difference will never go to 0 for any on-line algorithm but will be bounded from below at least by a constant.

Lueker [311] managed to close this gap between $\log^{3/2} n$ and a constant. He considered a variant of Threshold with objective function value z^{TL} where in iteration j the threshold T is set to the expected value of the efficiency of the split item of the remaining problem with capacity $c - W$ thus depending on the current filling of the knapsack in the way suggested above. While his contribution deals with general distributions it also implies that

$$E[z^* - z^{TL}] \text{ is in } O(\log n).$$

Roughly speaking this bound is tight in the sense that no on-line algorithm can improve upon this $\log n$ factor from an asymptotic point of view.

14.8.1 Time Dependent On-Line Knapsack Problems

In the intuition of the on-line knapsack model it is quite natural to think of a certain underlying time line. The items are somehow seen as arriving over time and a decision has to be made at the arrival time of an item before the next item arrives. Taking this time dependence explicitly into account several versions of on-line knapsack problems were investigated in a different setup and under a different name.

The *dynamic and stochastic knapsack problem* was treated extensively in several contributions by Papastavrou, Rajagopalan and Kleywegt [369] and Kleywegt and Papastavrou [276, 277]. Their model is based on the decision problem of a freight dispatcher in the transportation industry who receives a more or less random sequence of transportation requests each of which must be accepted or rejected without delay. In the former case a price is received and a resource consumed whereas in the latter case no resources are required but a negative profit due to the loss of costumer goodwill etc., may be incurred.

There is also a time line involved in the dynamic and stochastic knapsack problem, in particular the current filling of the knapsack will cause holding costs until the selection process is finished (which may be done by the decision maker at any time). In Papastavrou et al. [369], the timeline is finite meaning that after a given and known deadline is reached, no further requests arrive and no selections can be made.

In the model considered by Kleywegt and Papastavrou the items (i.e. freight orders) are assumed to arrive according to a Poisson process. All item profits are drawn from the same distribution which is known at the beginning of the process. In [276] it is assumed that all items require the same amount of the resource. This requirement is dropped in [277] where the item weights are also drawn from a common distribution. Note that the profits and weights are generated as pairs from a common two-dimensional distribution and allowed to be dependent.

The whole setup of the investigations for this problem is based on a Markov decision process. The time dependence hardly permits a comparison to the more classical situation of the previously discussed research on deterministic on-line knapsack problems. The achieved results contain general characterizations of optimal decision policies, expected optimal solution values and expected optimal stopping times for finite and infinite planning horizons. It turns out that the optimal policy is a memoryless *threshold policy* which can be seen as a generalization of the previous algorithm Threshold. Clearly, the value of T depends on the remaining capacity $c - W$ and on the time but also on additional details. Due to the dynamic nature of the problem most statements are rather complex recursive equations which we do not display here. These equations can be solved iteratively in the general case and in closed form for special distributions as illustrated in [276] for the exponential distribution.

A very similar model was applied independently by Van Slyke and Young [469] in the context of airline yield management. In this case the transportation requests correspond to airline customers who want to book a certain flight for a given prize. Yield management deals with the problem of an airline of accepting requests in such a way that a flight is not overbooked (in a perfect world, cf. O'Rourke [362]) and the overall revenue, i.e. the total sum of prizes paid by the customers, is as large as possible. Clearly, requests by single passengers would be modeled by setting $w_j = 1$, but group bookings require a model with general integer weights w_j.

As in [369] a strict deadline is given for the planning period by the take-off time of the flight. However, the main difference of the model in [469] to the previously discussed scenario is the fact that the arrival of requests is not constant over time. Indeed, the crucial feature of yield management is the fact that there is for many flights a larger number of non-commercial travelers, who tend to book early and look for a cheap prize, whereas the smaller number of business travelers tend to book late and can be charged higher prizes.

Van Slyke and Young [469] analyze the problem for requests arriving by a Poisson process with time dependent arrival rates. They derive general descriptions of packing policies which are optimal in expectation and apply these to the airline yield management problem for the single-leg case. By certain illustrative examples they can show that some natural conjectures on an optimal policy are not valid and that even simple examples can have complicated optimal policies. The more complex multihop case, which includes travel itineraries with stopovers, can be modeled by

the analogous stochastic version of a multidimensional knapsack problem (d-KP), which is also treated in [469].

Moving even further away from the classical knapsack model, a number of papers appeared under the title "stochastic knapsack problem" with a stronger notion of time dependence. These contributions originate from the area of telecommunication systems where requests for service such as transmission capacity arrive over time and also expire after their completion. This scenario can be modeled as a generalization of the dynamic and stochastic knapsack problem where the selection of an item causes the consumption of a certain amount of a bounded resource but only for a given time period. Introducing a discrete time line with d time steps, the deterministic version of the problem could be modeled as an instance of (d-KP) where an item j with resource demand w_j, which is present from time period s_j to e_j, has weight coefficients

$$w_{ij} := \begin{cases} 0 & \text{for } i = 1,\ldots,s_j - 1, \\ w_j & \text{for } i = s_j,\ldots,e_j, \\ 0 & \text{for } i = e_j + 1,\ldots,d. \end{cases}$$

In the stochastic model discussed in the literature the requests (i.e. the items) arrive by a stochastic process, usually a Poisson process, and also have stochastic durations. The seminal publication in this area focusing on the connection to the knapsack problem seems to be due to Ross and Tsang [410]. Since the resulting research is closer to queueing theory and stochastic processes (see e.g. the textbook by Ross [411]) than to the classical treatment of probabilistic aspects of the knapsack problem, we will not go into this area any further.

15. Some Selected Applications

In the preceding chapters of this book on the different variations of the knapsack problem we mentioned also examples for applications of the specific problem treated in a chapter. In this section we go into depth with some selected applications of the knapsack problem. Our intention is not to cover all possible kinds of applications but to present some selected examples which illustrate the bandwidth of the fields where knapsack problems appear. Among the problems we consider are cutting problems, column generation, separation of cover inequalities, financial decision problems, knapsack cryptosystems and combinatorial auctions.

15.1 Two-Dimensional Two-Stage Cutting Problems

The main ingredients of a *two-dimensional cutting problem* are a given set of rectangular pieces and a large rectangular sheet or stock unit which should be cut into pieces out of the given set. Industrial applications of cutting problems deal with various materials such as steel, wood and glass. In most cases the cutting technology permits only so-called *guillotine cuts*, i.e. every cut of the large sheet has to be made orthogonal to one edge and straight across the sheet from one side to the opposite side without stopping. Moreover, usually only axis parallel cuts are allowed.

It depends on the raw material whether it is allowed to choose the *orientation* of the pieces when they are cut from the large sheet. If the pieces have a given surface structure such as the grain of wood or the structure of tinted glass, every piece may have a fixed orientation prescribing a correspondence of the axes of the pieces with axes of the large sheet. In the case of homogeneous material such as steel or chipboard the pieces may be rotated by 90 degrees.

A further restriction caused either by the cutting technology or the desired ease of manipulation is the requirement to perform cuts in *stages*. This means that in a first stage the large sheet is cut vertically into a number of *strips*. In a second stage each strip is further cut horizontally into rectangular pieces. Further stages my follow to continue the cutting process but many real-world cases are restricted to *two-stage cutting patterns*. Since the pieces resulting from a two-stage cutting procedure may have larger dimensions than the required pieces, excess material

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

must be removed during a final processing step called *trimming*. Illustrations of two-stage cutting patterns are given in Figure 15.1 (with trimming) and in Figure 15.2 (without trimming). A detailed example of two-stage guillotine cutting in hardboard industry can be found in Morabito and Garcia [349].

From an optimization point of view there are two main tasks which will be considered in the following self-contained sections. The first one is a special case of the *two-dimensional bin packing problem* and consists of a set of pieces each of which carries a certain demand value indicating the number of required copies which must be cut. The objective is to minimize the number of large sheets which are necessary to cut all required pieces. A simple knapsack based heuristic for this problem will be described in Section 15.1.1.

The second scenario is the optimal utilization of one large sheet. This means that the set of rectangular pieces represents stock that may be produced but no demand is given for each piece. In this case the objective is to maximize the total profit (usually given as total area of the cut pieces) for one given sheet. This problem will be further discussed in Section 15.1.2.

There is a vast literature on two-dimensional cutting algorithms with contributions appearing in a wide range of different scientific outlets. Recent surveys were given by Lodi, Martello and Monaci [303] and Lodi, Martello and Vigo [305]. Computational comparisons of modern heuristics were performed by Alvarez-Valdes, Parajon and Tamarit [9]. In this section we will only point out some occurrences of knapsack problems in this context but refrain from given a full introduction to the topic. It should be noted that two-dimensional geometric packing problems are closely related, in fact often equivalent, to two-dimensional cutting problems. However, features like guillotine cuts and forbidden rotations do not carry over to the packing formulation.

15.1.1 Cutting a Given Demand from a Minimal Number of Sheets

We are given a set of n item types where type j has dimension $\ell_j \times w_j$ and a demand b_j. All $\sum_{j=1}^{n} b_j$ pieces must be produced by two-stage guillotine cuts without rotations from a minimal number of large sheets with dimension $L \times W$. It should be mentioned that even if the two-stage property, which is also known as *level-oriented packing*, is not required from the cutting pattern, many heuristics construct two-stage patterns to reduce the complexity of the problem. Recent publications for this two-stage problem are due to Lodi, Martello and Vigo [304] and Caprara, Lodi and Monaci [66]. An extension of the level-oriented approach in a heuristic for the three-dimensional packing of a single container is given by Pisinger [390].

The two-stage requirement leads directly to the following basic strategy described in [304]. After determining the item type with the maximum length given w.l.o.g. by $\ell_{\max} = \ell_1$, a first strip is cut with length $\ell_{\max}$. To determine the other item types

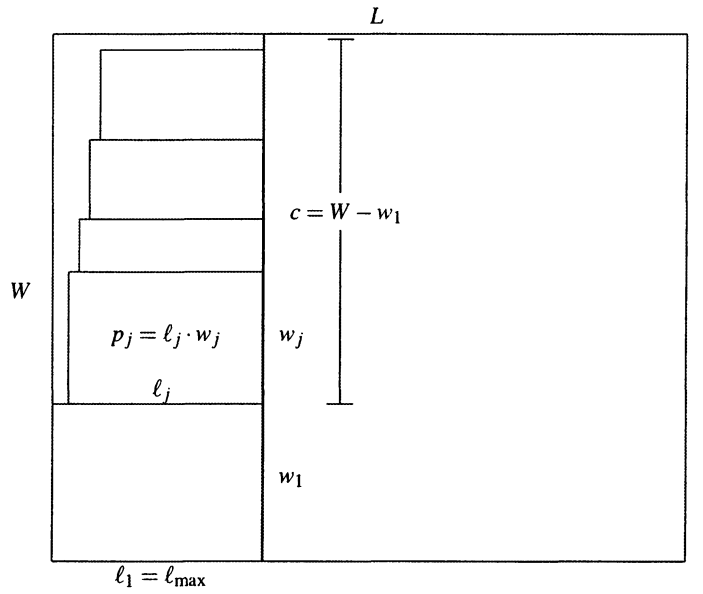


Fig. 15.1. First strip of length ℓ_1 filled by the solution of a bounded knapsack problem with capacity $c = W - w_1$.

to be cut from this strip different heuristic ideas based on one-dimensional bin packing were suggested but the locally best choice is given by the solution of a bounded knapsack problem. Every item type j gives rise to an item type j in the corresponding instance of (BKP) with profit $p_j = \ell_j w_j$ and weight w_j. The bound on the number of copies of item types in (BKP) is given by the number of remaining item types to be cut (b_j in the beginning). The capacity corresponding to the available width in the strip is given by $W - w_1$. An illustration of a strip packed by this approach is given in Figure 15.1. Since the arising instances of (BKP) are of moderate size, even optimal solutions can be computed for these subproblems in reasonable time.

The procedure is iterated in the obvious way by selecting the maximum length among all remaining item types to start the next strip and filling this strip by the solution of the corresponding instance of (BKP) for the set of remaining pieces. Continuing this process we end up with a set of feasible strips. It remains to combine these strips into sheets. This is equivalent to solving a classical one-dimensional bin packing problem where the length of a strip corresponds the size of an item and the L is the bin capacity.

By simply arranging the constructed strips in an arbitrary order, we get a solution of the related *two-dimensional strip packing problem*, where an infinite strip of width W is given instead of the finite sheets. In this variant of the problem the total length required to fulfill the total demand should be minimized. It appears in particular in

the cutting of steel coils and paper rolls (see e.g. Ferreira, Neves and Castro [440], and Valério de Carvalho and Guimarães Rodrigues [465]).

It is easy to adapt this heuristic to the problem where a rotation of the pieces is allowed. In this case, the length of a new strip is given by $\max\{\min\{\ell_j, w_j\} \mid j = 1,\ldots,n\}$ over all remaining item types. To achieve a dense filling of pieces in the (BKP) solution for the remainder of the strip the remaining item types are oriented with their longer side parallel to the length of the strip, if this side does not exceed the strip length. Otherwise, their shorter side, which by definition cannot exceed the strip length, will be oriented parallel to the length of the strip.

15.1.2 Optimal Utilization of a Single Sheet

This section deals with optimal utilization of one large rectangular sheet which should be cut into rectangular pieces as described in the beginning of Section 15.1. Each piece of the given set has an associated profit and the objective is to select those pieces for cutting which yield the maximum total profit. Since each piece may be produced arbitrarily many times the resulting two-dimensional cutting problem is a kind of an unbounded knapsack problem in two dimensions. In this section we assume again that only two-stage guillotine cuts are performed and no rotations are allowed.

Dynamic programming algorithms for the two-stage cutting problem were presented by Gilmore and Gomory [174, 175, 176]. More recent papers on the topic include Valério de Carvalho and Guimarães Rodrigues [464], Fayard and Zissimopoulos [140], Hifi and Zissimopoulos [226], and Hifi [223]. An exact algorithm for the non-guillotine case which includes the solution of a standard knapsack problem as subroutine was given by Hadjiconstantinou and Christofides [204].

In order to formulate the problem as an integer programming model, we repeat some definitions: The large sheet has dimension $L \times W$. There are n different piece types, each having dimension $\ell_j \times w_j$ which can be produced in unbounded numbers. The pieces have an associated profit p_j reflecting some demand prices. If we set $p_j = \ell_j w_j$ then the objective is to use the maximal area of the rectangular sheet. For the integer programming formulation an upper bound k for the number of vertical strips is required. A trivial bound can be given by $k = \lfloor L/\ell_{\min} \rfloor$. Now the lengths of these vertical strips can be represented by variables y_i for $i = 1,\ldots,k$. We use the decision variable x_{ij} to represent that item j is cut x_{ij} times from strip i. The decision variable x'_{ij} says that at least one item j has been cut from strip i. Moreover, let m_j denote the maximum number of pieces of type j which can be cut from the sheet. An example of a two-stage cutting pattern is given in Figure 15.2.

The integer-programming model now becomes

$$z = \text{maximize} \sum_{i=1}^{k} \sum_{j=1}^{n} p_j x_{ij} \tag{15.1}$$

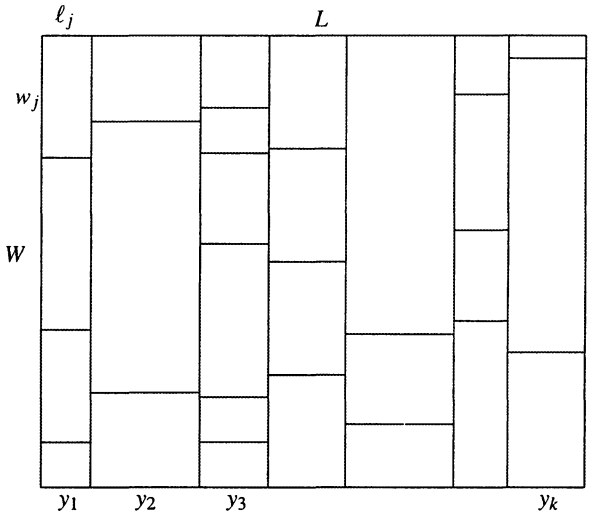


Fig. 15.2. A two-stage cutting pattern.

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_{ij} \le W, \quad i = 1,\dots,k, \tag{15.2}$$

$$\sum_{i=1}^{k} y_i \le L, \tag{15.3}$$

$$\ell_j x'_{ij} \le y_i, \quad i = 1,\dots,k,\ j = 1,\dots,n, \tag{15.4}$$

$$m_j x'_{ij} \ge x_{ij}, \quad i = 1,\dots,k,\ j = 1,\dots,n, \tag{15.5}$$

$$x_{ij} \in \{0,\dots,m_j\}, \quad i = 1,\dots,k,\ j = 1,\dots,n,$$

$$x'_{ij} \in \{0,1\}, \quad i = 1,\dots,k,\ j = 1,\dots,n,$$

$$y_i \in \{1,\dots,L\}, \quad i = 1,\dots,k.$$

Here, the constraints (15.2) say that no strip may exceed the width of the sheet, the constraint (15.3) ensures that the sum of strip lengths should be within the sheet length, while the constraints (15.4) demand that each item j chosen for strip i should have length ℓ_j not greater than the length y_i of the strip. The constraint (15.5) ensures that if $x_{ij} > 0$ then $x'_{ij} = 1$.

W.l.o.g. we may assume that $y_1 \le y_2 \le \dots \le y_k$ since we may cut the strips in any order.

The problem may be solved in two iterations. First, for each strip length $y_i = 1,\dots,L$ we find the maximal obtainable profit sum in a strip of width W by solving the following problem defined for the set of item types with length $\ell_j \le y_i$:

$$\begin{aligned} \overline{p}_i = \text{maximize} & \sum_{j=1}^{n} p_j x_{ij} \\ \text{subject to} & \sum_{j=1}^{n} w_j x_{ij} \le W, \\ & x_{ij} \in \{0,\dots,m_j\},\ j \in \{1,\dots,n\} \text{ with } \ell_j \le y_i. \end{aligned} \tag{15.6}$$

Note that this optimization problem is closely related to the instances of (BKP) defined in Section 15.1.1 for cutting a single strip. Next, setting $\lambda_i := y_i$ for $i = 1,\dots,L$ we determine the most profitable combination of strips which fills out the sheet length L.

$$\begin{aligned} \text{maximize } z = & \sum_{i=1}^{n} \overline{p}_i x_i \\ \text{subject to} & \sum_{i=1}^{n} \lambda_i x_i \le L, \\ & x_i \in \{0,\dots,L\},\ i = 1,\dots,L. \end{aligned} \tag{15.7}$$

Now the solution to (15.7) is the optimal solution to (15.1).

Gilmore and Gomory noticed that the above problems can be solved to optimality in pseudopolynomial time by solving *only two* unbounded knapsack problems. The idea is to order the items according to increasing lengths ℓ_j and solve problem (15.6) through dynamic programming. In this way, all possible strip lengths $y_i = \ell_j$ are taken into account. (It does not make sense to cut a strip with length not equal to any item length.) For $d = 0,\dots,W$ and $j = 1,\dots,n$ let $z_j(d)$ be defined as

$$z_j(d) := \max\left\{ \sum_{k=1}^{j} p_k x_k \,\middle|\, \sum_{k=1}^{j} w_k x_k \le d, x_k \in \mathbb{N}_0, k = 1,\dots,j \right\}. \tag{15.8}$$

Using recursion (8.14) from Section 8.3, we may find all the above solutions in $O(nW)$ time. Due to the ordering of the items, $z_j(W)$ will be an optimal solution to (15.6). Problem (15.7) is recognized as an unbounded knapsack problem (UKP) and hence may be solved in $O(nL)$ time through the same recursion, since we have only n distinct values of $\overline{p}_i$.

A further generalization of the problem would be to allow rotation of the items by 90 degrees. In this case we may simply extend the problem to $2n$ items, item $n + j$ being item j rotated. Since an infinite amount of both items is available, the above recursion may be applied without changes.

If the size $L \times W$ of the available sheet may be chosen arbitrarily, we may use the same algorithm several times, as the dynamic programming algorithm solves (15.6) for all sheet widths W, and thus only the given length should be tested in (15.7).

15.2 Column Generation in Cutting Stock Problems

The technique of column generation is a method to solve relatively quickly linear programming problems with a huge number of variables compared to the number of constraints. Column generation works without explicitly generating the whole constraint matrix. It exploits the fact that the number of basic variables, and thus nonzero variables, in an optimal solution is equal to the number of constraints and so, for constructing an optimal solution we need only a few columns of the constraint matrix. In principle, column generation solves a series of dual problems with a smaller number of variables by iteratively increasing the variables of a master problem.

The classical example for column generation is the cutting stock problem where the dual problems mentioned above turn into knapsack problems. In the *cutting stock problem* we are given an unlimited number of bins (e.g. base rolls of metal) of length c and m different types of items (rolls). At least b_i rolls of length w_i, $i = 1,\ldots,m$ have to be cut of the base rolls. The objective is minimize the the number of bins used, i.e. to minimize the wasted material. In the special case of $b_i = 1$, $i = 1,\ldots,m$, i.e. there is only a demand for one item per type, the problem can be recognized as the ordinary *bin packing problem.*

Assume that U is an upper bound on the number of bins needed. Moreover, the variable x_{ij} denotes how many times item type i is cut of bin j and the decision variable y_j denotes whether bin j is used for cutting or not. The cutting stock problem may then be formulated as a follows:

$$\begin{aligned}
&\text{minimize } \sum_{j=1}^{U} y_j \\
&\text{subject to } \sum_{i=1}^{n} w_i x_{ij} \leq c y_j, \quad j = 1,\ldots,U, \\
&\sum_{j=1}^{U} x_{ij} \geq b_i, \quad i = 1,\ldots,m, \\
&x_{ij} \in \mathbb{N}_0, y_j \in \{0,1\}, \quad i = 1,\ldots,m, j = 1,\ldots,U.
\end{aligned} \tag{15.9}$$

Unfortunately, the upper bounds obtained by solving the LP-relaxation of (15.9) may be quite loose and the problem contains many symmetric solutions (see also the paper by Vance et al. [470]). Thus, problem (15.9) is extremely difficult for general mixed integer programming solvers and this model is not used in practice.

A different formulation of the cutting stock problem is due to Gilmore and Gomory [174, 175]. Instead of assigning an item to the bin it is cut of, we consider the set all possible *cutting patterns*. A vector $\alpha \in \{0,1\}^m$ represents a cutting pattern if we have

$$\sum_{i=1}^{m} w_i \alpha_i \leq c. \tag{15.10}$$

where α_i denotes how often item type i is cut in the corresponding pattern. Let the matrix $A = (a_{ij})$ be an $m \times n$ matrix which consists of the n possible packing patterns j of a single bin, i.e. each column j in A should satisfy (15.10). Then we may formulate the cutting stock problem as choosing a subset of the columns (i.e. packing patterns) such that the demands are satisfied. If we introduce the decision variable x_j to denote how many times a packing pattern j is used, we get the model

$$\begin{aligned} \text{minimize } & \sum_{j=1}^{n} x_j \\ \text{subject to } & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = 1,\ldots,m, \\ & x_j \in \mathbb{N}_0, \quad j = 1,\ldots,n. \end{aligned} \tag{15.11}$$

Here the objective function minimizes the number of bins used and the constraints guarantee that each required item is cut of one bin. If the demands b_i are not too small, we can expect the LP-relaxation of (15.11) to deliver reasonable results for the cutting stock problem after simply rounding up the noninteger pattern numbers x_j. Unfortunately, the number of possible packing patterns can be exponentially large. So, it may be beneficial to avoid formulating the whole matrix A explicitly.

In order to solve the LP-relaxation of the so-called *master problem* (15.11) one may use *column generation* to introduce packing patterns gradually as they are needed by the simplex algorithm. For explaining the principle of column generation consider the matrix formulation of a linear program in equality form (including slack variables) with A an $m \times (m+n)$ matrix, $x, c \in \mathbb{R}^{m+n}$ and $b \in \mathbb{R}^m$.

$$\begin{aligned} \text{minimize } & c^T x \\ \text{subject to } & Ax = b, \\ & x \geq 0. \end{aligned} \tag{15.12}$$

W.l.o.g. assume that a basic solution $x = (x_B, x_N)$ where x_B represents the basic variables and x_N the nonbasic variables. Matrix A can be represented as $A = (A_B, A_N)$ with A_B a nonsingular $m \times m$ matrix and A_N a $m \times n$ matrix. In this way the constraint of (15.12) can be written as $A_B x_B + A_N x_N = b$. Expressing the basic variables in the objective function by the nonbasic variables, (15.12) turns into

$$\begin{aligned} \text{minimize } & (c_N - c_B A_B^{-1} A_N) x_N + c_B A_B^{-1} b \\ \text{subject to } & x_B = A_B^{-1} b - A_B^{-1} A_N x_N, \\ & x \geq 0. \end{aligned} \tag{15.13}$$

It is well-known, that a basic feasible solution is optimal if the reduced costs are all nonnegative, i.e.

$$c_N - c_B A_B^{-1} A_N \geq 0. \tag{15.14}$$

In the case of the cutting stock problem we have $c = (1,\ldots,1)$ due to the definition of the objective function in (15.11). Setting $y := (c_B A_B^{-1})$, the values y_i, $i = 1,\ldots,m$, represent in fact the values of the dual variables in an optimal solution of (15.11). Let A_j denote the j-th column of A_N, $j = 1,\ldots,n$. Then the optimality condition (15.14) for the cutting stock problem turns into

$$(15.14) \Longleftrightarrow (1 - yA_j) \geq 0 \text{ for } j = 1,\ldots,n \Longleftrightarrow \max_{j=1,\ldots,n} yA_j \leq 1.$$

This must hold for all cutting patterns A_j. Hence we maximize the reduced cost coefficients over an unknown pattern α. Thus, a given basic solution x for (15.11) is optimal if and only if the optimal solution of the knapsack problem

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{m} y_i \alpha_i \\ &\text{subject to} \sum_{i=1}^{m} w_i \alpha_i \leq c, \\ &\qquad \alpha_i \in \{0,1\}, \quad i = 1,\ldots,m, \end{aligned} \tag{15.15}$$

is less than or equal to one. The knapsack problem (15.15) is a pricing problem which looks for the column with the most negative reduced cost coefficient. According to (15.10) the capacity constraint guarantees that the values α_i form a cutting pattern.

The *restricted master problem* is the master problem (15.11) initialized with a small subset of the columns of A, i.e. reducing A to a small subset of columns by setting the variables corresponding to the other columns to zero. Now the overall principle of column generation can be described as follows:

1. Find an easy initial feasible solution for the master problem, where only a few variables are nonzero, e.g. only one type of items is cut of each bin. The corresponding packing patterns form the initial columns of A in the reduced master problem.
2. Solve the reduced master problem for the current value of A and let y_i be the corresponding dual variables.
3. Solve a knapsack problem of the form (15.15). If the optimal solution value is smaller than 1, then stop. We have found an optimal solution of the master problem.
4. Add the computed knapsack solution $\alpha = (\alpha_1,\ldots,\alpha_m)$ as a new column to the matrix A and go to Step 2.

Example: Consider an instance with the following demands: 4 items of length 3, 3 items of length 5 and 2 items of length 6. All the items have to be cut of bins of length 10. Obviously, no more than 6 bins are necessary to cut all items. Hence using the first model (15.9), the problem may be formulated as the following integer linear model:

$$
\begin{aligned}
\text{minimize } & y_1 + y_2 + y_3 + y_4 + y_5 + y_6 \\
\text{subject to } & 3x_{11} + 5x_{12} + 6x_{13} \leq 10y_1, \\
& 3x_{21} + 5x_{22} + 6x_{23} \leq 10y_2, \\
& 3x_{31} + 5x_{32} + 6x_{33} \leq 10y_3, \\
& 3x_{41} + 5x_{42} + 6x_{43} \leq 10y_4, \\
& 3x_{51} + 5x_{52} + 6x_{53} \leq 10y_5, \\
& 3x_{61} + 5x_{62} + 6x_{63} \leq 10y_6, \\
& x_{11} + x_{21} + x_{31} + x_{41} + x_{51} + x_{61} \geq 4, \\
& x_{12} + x_{22} + x_{32} + x_{42} + x_{52} + x_{62} \geq 3, \\
& x_{13} + x_{23} + x_{33} + x_{43} + x_{53} + x_{63} \geq 2, \\
& x_{ij} \in \mathbb{N}_0, \quad y_j \in \{0,1\}.
\end{aligned}
$$

The optimal solution of the LP-relaxation is $y_1 = 1, y_2 = \frac{1}{5}, y_3 = 1, y_4 = 1, y_5 = \frac{1}{5}$, $y_6 = \frac{1}{2}$ with objective value $3\frac{9}{10}$. Since the optimal integer programming solution value is 5, the bound from the LP-relaxation is quite bad.

A complete formulation of the master problem based on (15.11) using the packing patterns (3,0,0), (0,2,0), (0,0,1), (1,1,0), (1,0,1) is given by

$$
\begin{aligned}
\text{minimize } & x_1 + x_2 + x_3 + x_4 + x_5 \\
\text{subject to } & 3x_1 + 1x_4 + 1x_5 \geq 4, \\
& 2x_2 + 1x_4 \geq 3, \\
& 1x_3 + 1x_5 \geq 2, \\
& x_j \in \mathbb{N}_0, \quad j = 1,\ldots,5.
\end{aligned}
$$

with optimal solution values $x_1 = 1, x_2 = 2, x_3 = 1, x_4 = 0, x_5 = 1$.

Since the number of columns involved in general may be quite huge, we use column generation to solve the LP-relaxation of the above model. Initially we generate cutting patterns where only one item type is cut of each bin. This gives the initial reduced master problem

$$
\begin{aligned}
\text{minimize } & x_1 + x_2 + x_3 \\
\text{subject to } & 3x_1 \geq 4, \\
& 2x_2 \geq 3, \\
& 1x_3 \geq 2, \\
& x_1, x_2, x_3 \geq 0.
\end{aligned}
$$

Solving the LP-solution we get the dual variables $y_1 = \frac{1}{3}, y_2 = \frac{1}{2}, y_3 = 1$. This leads to the pricing problem (15.15)

$$\begin{aligned} \text{maximize } & \tfrac{1}{3}\alpha_1 + \tfrac{1}{2}\alpha_2 + 1\alpha_3 \\ \text{subject to } & 3\alpha_1 + 5\alpha_2 + 6\alpha_3 \leq 10, \\ & \alpha_1, \alpha_2, \alpha_3 \in \mathbb{N}_0 \end{aligned}$$

with solution $\alpha_1 = 1$, $\alpha_2 = 0$, $\alpha_3 = 1$. Adding the resulting column to the reduced master problem we find

$$\begin{aligned} \text{minimize } & x_1 + x_2 + x_3 + x_4 \\ \text{subject to } & 3x_1 + x_4 \geq 4, \\ & 2x_2 \geq 3, \\ & 1x_3 + x_4 \geq 2, \\ & x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

The optimal dual variables of this problem are $y_1 = \frac{1}{3}$, $y_2 = \frac{1}{2}$, $y_3 = \frac{2}{3}$. This leads to the pricing problem

$$\begin{aligned} \text{maximize } & \tfrac{1}{3}\alpha_1 + \tfrac{1}{2}\alpha_2 + \tfrac{2}{3}\alpha_3 \\ \text{subject to } & 3\alpha_1 + 5\alpha_2 + 6\alpha_3 \leq 10, \\ & \alpha_1, \alpha_2, \alpha_3 \in \mathbb{N}_0 \end{aligned}$$

with solution $\alpha_1 = 1$, $\alpha_2 = 0$, $\alpha_3 = 1$. Since the objective value is 1 we may terminate. The objective of the master problem is $4\frac{1}{6}$ with $x_1 = \frac{2}{3}$, $x_2 = \frac{3}{2}$, $x_4 = 2$. The LP-solution may be rounded up to 5 which also is the optimal solution value. □

15.3 Separation of Cover Inequalities

Cover inequalities have been introduced in Section 3.10 and 5.1.1. To recapitulate, the *knapsack polytope* P is given by $P := \text{conv}\{x \in \mathbb{R}^n \mid \sum_{j=1}^n w_j x_j \leq c, x_j \in \{0,1\}, j = 1,\ldots,n\}$. A subset $S \subseteq N$ is a *cover* for P if $\sum_{j \in S} w_j > c$. Moreover, it is a *minimal cover* for P if $\sum_{j \in S \setminus \{i\}} w_j \leq c$ for all $i \in S$. For a minimal cover S, the corresponding *cover inequality* given by $\sum_{j \in S} x_j \leq |S| - 1$ is a valid inequality for P. Let $\mathcal{F}$ be the family of cover inequalities for P.

A natural question is, whether it is possible to automatically generate these inequalities for a given instance, i.e. to solve the so-called *separation problem* for the family $\mathcal{F}$. The following algorithm was presented by e.g. Balas [18] and Wolsey [488].

Consider a binary integer programming model which contains the knapsack inequality

$$\sum_{j=1}^{n} w_j x_j \leq c \tag{15.16}$$

where we assume that all w_j are nonnegative integers. Let the solution of the LP-relaxation of the problem be denoted as $x^{LP} = (x_1^{LP}, \ldots, x_n^{LP})$. If all x_j^{LP} are integers in the constraint (15.16) we are done. Otherwise we are interested in knowing whether

x^{LP} satisfies all the inequalities in $\mathcal{F}$, and if this is not the case, to separate the most violated cover inequality. For this purpose we solve the problem

$$\begin{aligned} z_{\min} = \text{minimize } & \sum_{j=1}^{n}(1-x_j^{LP})y_j \\ \text{subject to } & \sum_{j=1}^{n} w_j y_j \geq c+1, \\ & y_j \in \{0,1\}, \quad j=1,\ldots,n. \end{aligned} \tag{15.17}$$

which is a *minimization knapsack problem* which has been presented in Section 13.3.3. The coefficients$(1-x_j^{LP})$ in the objective are real values, but since the coefficients w_j in the capacity constraint are integers, the problem may be solved by use of dynamic programming. Notice that the coefficients satisfy that $0 \leq 1-x_j^{LP} \leq 1$, and when $1-x_j^{LP}=0$, we may fix y_j to zero.

If we obtain a solution to (15.17) where $z_{\min} < 1$ we may construct a cover inequality as follows. Let

$$C := \{j \in N | y_j = 1\} \tag{15.18}$$

be the set of items chosen in (15.17). Then the most violated cover inequality of (15.16) is given by

$$\sum_{j\in C} x_j \leq |C| - 1. \tag{15.19}$$

Notice that

$$\sum_{j\in C} w_j = \sum_{j=1}^{n} w_j x_j \geq c+1 > c,$$

hence C is a cover. It is a *minimal* cover, since if we were able to remove an item j from C and still have a cover, we would have a solution to (15.17) with a smaller objective function. It is the most *violated* inequality since the violation of (15.19) for a given cover C is given by

$$|C| - 1 - \sum_{j\in C} x_j = \sum_{j\in C}(1-x_j) - 1$$

which is maximized by the objective function in (15.17).

Example: Consider the following integer programming model with a single constraint:

$$\begin{aligned} \text{maximize } & 8x_1 + 10x_2 + 6x_3 + 12x_4 + 8x_5 \\ \text{subject to } & 4x_1 + 6x_2 + 2x_3 + 8x_4 + 4x_5 \leq 8, \\ & x_j \in \{0,1\}, \quad j=1,\ldots,5. \end{aligned} \tag{15.20}$$

The LP-solution is $x_1^{LP}=1, x_3^{LP}=1, x_5^{LP}=\frac{1}{2}, x_2^{LP}=x_4^{LP}=0$ with objective value 18. As the optimal integer solution is $x_1=1, x_5=1$ with objective value 16, the

bound from LP-relaxation is quite bad, and hence it may be beneficial to tighten the formulation by adding valid inequalities.

To separate the most violated inequality we solve (15.17) which in our case becomes

$$\begin{aligned} z_{\min} = \text{minimize } & 0y_1 + 1y_2 + 0y_3 + 1y_4 + \tfrac{1}{2}y_5 \\ \text{subject to } & 4y_1 + 6y_2 + 2y_3 + 8y_4 + 4y_5 \geq 9, \\ & y_j \in \{0,1\}, \quad j = 1,ldots,5, \end{aligned}$$

having the solution $y_1 = 1, y_3 = 1, y_5 = 1$ with all other variables equal to 0. Since $z_{\min} = \frac{1}{2} < 1$ a violated inequality is given by (15.19) with $C = \{1,3,5\}$. Adding the cover inequality to the problem (15.20) we get the tighter formulation

$$\begin{aligned} \text{maximize } & 8x_1 + 10x_2 + 6x_3 + 12x_4 + 8x_5 \\ \text{subject to } & 4x_1 + 6x_2 + 2x_3 + 8x_4 + 4x_5 \leq 8, \\ & x_1 + x_3 + x_5 \leq 2 \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,5. \end{aligned}$$

Solving the LP-relaxation of the latter problem we get the solution $x_1^{LP} = 1, x_2^{LP} = \frac{1}{3}$, $x_3^{LP} = 1, x_4^{LP} = x_5^{LP} = 0$ with objective value $17\frac{1}{3}$.

We can now repeat the above process to separate another cover inequality. □

15.4 Financial Decision Problems

In this section occurrences of knapsack problems in the context of economics, in particular in finance, are presented. We start with classical capital budgeting problems which seem to be the earliest applications of the knapsack problem in the literature. More specific problems arising in the granting of loans and the interbank payment processing are presented in Sections 15.4.2 and 15.4.3. A more complex decision problem in the area of securitization will be described in Section 15.5.

15.4.1 Capital Budgeting

The optimal selection of investments was among the earliest problems modeled as a knapsack problem. It is also one of the standard motivations for the introduction of the knapsack problem in a classroom. Vintage publications such as Lorie and Savage [306] from 1955 and Everett [135] in 1963, the slightly less ancient papers by Weingartner and Ness [479] and Nemhauser and Ullmann [358], and the early survey by Weingartner [478] considered variants of profit optimization where certain projects resp. investment opportunities may be selected (a binary choice) subject to a budget constraint. Each project has a certain *present value* which is usually calculated as the discounted sum of future returns minus the initial investment. Also

projects where several payments by the investor are necessary during a given time horizon, or where reinvesting and borrowing is possible, can be evaluated in this way (cf. [358]). The formulation of this basic problem as an instance of (KP) is obvious.

Many publication in the area of knapsack problems, in particular for (d-KP), seek motivation from capital budgeting problems but usually refrain from going into details. In the following we will briefly describe some more elaborate cases of economic scenarios with knapsack-type formulations. These and others were also described by Beinsen and Pferschy [31].

The classical decision problem of a government agency on a local or federal level is the allocation of funds for projects usually on a yearly basis. Typically, every department or interest group submits a long list of projects with their costs w_j. Obviously, it will rarely be the case that all these "wishes" can be fulfilled. Instead, it has to be decided which of the projects can actually be implemented with the available budget c. Each project is evaluated for its general *utility* or *benefit*. As far as possible a quantitative estimate p_j for this benefit is sought. Maximizing the total benefit while keeping total expenditures below the available budget immediately yields an instance of (KP). Some projects may be implemented in a number of more or less identical copies (e.g. kindergarten, primary school, vehicles for transportation or municipal duties), which results in an instance of a bounded or unbounded knapsack problem (BKP) or (UKP).

This simplified point of view can be brought closer to reality by considering the situation where the departments by wise foresight have refined their list of projects and include for every project several versions, e.g. a "full-size version", a reasonable "basic-version" and a trimmed down "economy-version". Now the decision process can be modeled as a multiple choice knapsack problem (MCKP) as discussed in Chapter 11. Exactly one of the variants of every project has to be selected, i.e. every class of items N_j corresponds to one project. To allow for the complete rejection of a project a "dummy-version" corresponding to the total cancellation of the project must be introduced.

15.4.2 Portfolio Selection

A different decision problem is encountered by a financial institution such as a commercial bank, which gives credit to its customers. In periods of low liquidity of the bank, credit will be rationed and assigned to costumers such that certain expectations on profit and risk are met (cf. [31]).

Each individual credit manager is assigned a certain credit volume C to be issued to costumers. Having received a collection of credit applications $1,\ldots,n$, each with a credit volume c_j, a certain creditworthiness or risk grade g_j is assigned to each application depending on the economic situation of the costumer, the higher the grade g_j, the lower the level of risk. In addition, an expected profit p_j to be gained

from every approved credit contract is determined. Naturally, this profit is dependent on the credit volume c_j and strongly inverse correlated to the credit standing g_j.

In general each costumer is seeking a fixed credit volume and is not interested in only a partial approval. Thus the credit manager has to decide which of the credit applications to approve such that the total credit volume does not exceed C and the total risk exposure measured by the sum of c_j/g_j is below an acceptable level L imposed by a higher level of management.

To maximize the overall gain for the bank, the task of the credit manager is a very simple version of a classical portfolio selection problem and can be modeled as a 2-dimensional knapsack problem (2-KP), with $x_j \in \{0,1\}$ indicating whether or not credit is granted to costumer j.

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} c_j x_j \leq C, \\ & \sum_{j=1}^{n} \frac{c_j}{g_j} x_j \leq L, \\ & x_j \in \{0,1\},\ j = 1,\ldots,n. \end{aligned} \tag{15.21}$$

In practice, it will be in most cases impossible to assign the exact amount of capital C to the customers. To include the "left-over" capital in the model the possibility of a risk-free investment is considered by introducing a virtual customer $n+1$ with $x_{n+1} \in \mathbb{N}_0$ who is willing to "accept" any integer multiple of $c_{n+1} = 1$ of credit with a risk factor $g_{n+1} = \infty$, i.e. $c_{n+1}/g_{n+1} = 0$, and a low profit of p_{n+1} following e.g. from the interbank rate. Thus, we attain the unbounded version of the two-dimensional knapsack problem. In this case (15.21) can be written as an equality.

The roles of risk and profit can also be reversed by imposing a minimum total profit value P and minimizing the total risk exposure. The resulting situation is related to the classical Markowitz model (see e.g. the textbook by Elton and Gruber [130]), however with a simplified linear (instead of quadratic) risk function and binary variables. It can be written as:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n+1} \frac{c_j}{g_j} x_j \\ \text{subject to} \quad & \sum_{j=1}^{n+1} p_j x_j \geq P, \\ & \sum_{j=1}^{n+1} c_j x_j = C, \end{aligned}$$

$$x_j \in \{0,1\},\ j = 1,\ldots,n,\ x_{n+1} \in \mathbb{N}_0.$$

Equivalently, it can be modeled as maximizing the risk of the rejected credit applications. Indeed, introducing the transformation $y_j := 1 - x_j$, $j = 1,\ldots,n$ and $y_{n+1} := C - x_{n+1}$ we get again a knapsack type problem with two constraints.

$$\begin{aligned}
\text{maximize } & \sum_{j=1}^{n+1} \frac{c_j}{g_j} y_j \\
\text{subject to } & \sum_{j=1}^{n+1} p_j y_j \leq \sum_{j=1}^{n} p_j + C p_{n+1} - P, \\
& \sum_{j=1}^{n+1} c_j y_j = \sum_{j=1}^{n} c_j + C(c_{n+1} - 1), \\
& y_j \in \{0,1\},\ j = 1,\ldots,n,\ y_{n+1} \in \mathbb{N}_0.
\end{aligned}$$

15.4.3 Interbank Clearing Systems

An interesting problem that most of us come into contact with in our personal financial life without giving it any thought, is the settlement of payments between banks. Every day a huge amount of money transfers takes place between individuals and companies. Naturally, in many cases the sender and the recipient of a payment will have their accounts with different banks. To avoid that every bank has to deal directly with all other banks to perform the corresponding payments, a *clearing house* collects all payment orders. It debits the sum of originated payments and credits the sum of received payments for each bank, and also organizes the data transfer.

While this zero-sum game would work perfectly well in a group of friends, a problem occurs if we are not sure whether all friends are able to fulfill their payment obligations or are faced with the danger of insolvency. Since the amounts of money that are dealt with are huge, even a very small risk of insolvency of one of the banks poses a significant problem for the other participants. Therefore, every bank participating in a clearing system has to deposit a fund to cover the execution of its payment orders as described in Güntzer, Jungnickel and Leclerc [202]. The payments are processed iteratively in a number of clearing runs. In every clearing run, the difference between outgoing payments and incoming payments for each participating bank must be smaller than its deposit. To improve the performance of the clearing house, the number of runs should be as small as possible. This can be achieved by maximizing in every run the overall clearing volume subject to the "overdraft" constraints following from the deposit condition.

Based on the largest German interbank clearing system, EAF-2 of the Landeszentralbank Hessen, a branch of the German central bank (Deutsche Bundesbank), a more detailed description of the whole operation and a formulation of the optimization problem was given by Güntzer, Jungnickel and Leclerc [202]. In their model of n banks, there are ℓ_{ij} payment orders from bank i to bank j submitted, each of them over an amount p_{ijk} for $k = 1,\ldots,\ell_{ij}$. For notational convenience we set $\ell_{ii} = 0$ for $i = 1,\ldots,n$. Denoting the deposit of bank i by c_i the problem of maximizing the overall clearing volume is modeled with binary variables x_{ijk} corresponding to the execution of the kth payment from bank i to j as follows.

$$\text{maximize} \quad \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{\ell_{ij}} p_{ijk}x_{ijk} \tag{15.22}$$

$$\text{subject to} \quad \sum_{j=1}^{n} \underbrace{\left(\sum_{k=1}^{\ell_{ij}} p_{ijk}x_{ijk} - \sum_{k=1}^{\ell_{ji}} p_{jik}x_{jik}\right)}_{\text{transfer from bank } i \text{ to bank } j} \leq c_i, \quad i = 1,\ldots,n,$$

$$x_{ijk} \in \{0,1\}, \quad i,j = 1,\ldots,n, k = 1,\ldots,\ell_{ij}.$$

Obviously, this optimization model corresponds to a multidimensional knapsack problem (d-KP), however with positive and negative item weights and an enormous number of variables.

To simplify the clearing system, the execution is split into two phases. The system starts with a bilateral phase where problem (15.22) is considered for $n = 2$, i.e. only the payments between pairs of banks are performed. In the following multilateral phase all n banks and all remaining payments are taken into account. Because of the huge number of variables, the authors of [202] concentrate on the development of simple approximation algorithms. These are based on primal and dual greedy-type methods (see Section 9.5.1).

Computational experiments showed that the methods used in practice could be improved significantly by the developed heuristics.

15.5 Asset-Backed Securitization

A rather sophisticated financial tool which has evolved only during the last twelve or fifteen years is the asset-backed securitization of financial assets, especially in the area of leasing contracts. In the following we will give a description of the underlying financial model and the resulting application of algorithms from the knapsack family. However, we will not go into the full details of this complex financial tool. The presentation follows to some extent the recent work by Mansini and Pferschy [317] based on the real-world situation of an Italian bank.

15.5.1 Introducing Securitization and Amortization Variants

Asset-Backed Securitization (ABS) is a financial tool which allows financial institutions (usually commercial banks) to move unmarketable assets (e.g. *lease assets*, *mortgage assets* or *commercial papers*) from their balance sheets in exchange for a long term loan which can be ploughed back into more profitable investments.

The (ABS) involves selling assets to a *special purpose vehicle* (SPV), an institution created solely for that purpose. The SPV funds the purchase through issuing debt securities (the notes) which are collateralized by the assets. The final investors who buy the notes receive periodic inflows (interests on their investments) which are directly related to the periodic installments paid by the holders of the assets (e.g. lessees or mortgage holders) to the originator (e.g. the lessor). Using the (ABS) structure the originator bypasses the problem of an impossible outright sale of its assets and thus reduces its overall exposure to such assets. This replacement of illiquid assets by marketable securities improves the return on equity (ROE).

From the strictly financial point of view of the originator, who will be the focus of attention in the following description, an (ABS) aims at three main objectives:

1. Replacement of the assets in its balance sheet, improving its ROE and allowing (if the originator is a bank) a more flexible observation of the constraints on the asset/liability composition imposed by the control authorities (i.e. the central bank).
2. Diversification of its funds sources. Although the originator may be low rated, the notes usually get a higher rating due to the bank and insurance companies which guarantee the whole operation. This implies that such notes can be traded on the main financial markets allowing the originator to get into markets which are usually forbidden to it, since attended only by more reliable companies.
3. Higher rated notes are more reliable investments and thus are allowed to pay lower interest rates to holders. If the cost to get higher rating is lower than the saving obtained by issuing notes with higher rating, then the global cost to acquire funds decreases.

The interest in this financial operation drastically increased in the last years all over Europe. Transactions of this type took place in particular in the area of public housing agencies and in the management of social security systems (cf. [317]). For a better insight in the complex problem of securitization we suggest textbooks such as Henderson and Scott [221] or Norton and Spellman [361]. Further references about theoretical contributions to (ABS) can be found e.g. in Mansini and Speranza [318].

In the following we will concentrate on an optimization problem of the originator introduced by Mansini and Speranza [318]. Having received the long term loan from the SPV the originator has to decide which assets to select for the transformation into securities. In particular, the assets have to be used to reimburse the loan in a

way such that the sum of outstanding principals of the selected assets never exceeds the outstanding principal of the main loan at any point in time. On the other hand the gap between the outstanding principal of the received loan and the outstanding principal of the selected assets should be minimized over all points in time. This gap constitutes a loss of profit due to missing more profitable investments with higher yields.

Naturally, the outstanding principal of an asset depends on the rule used for amortization. The case where the overall installment (sum of periodic interest and principal installment) is constant over time is known as *French amortization*. This has the convenient characteristic to require from the customers who hold assets (mortgage or lease contracts) the payment of the same amount at each deadline. The principal installments increase geometrically over time.

Mansini and Speranza [318] considered the case of *lease assets* where the outstanding principal is computed according to French amortization. The resulting problem of selecting assets *at one single point in time* can be modeled as a multidimensional knapsack problem (d-KP) as introduced in Section 1.2 and extensively treated in Chapter 9. For each reimbursement date of the main loan a constraint is introduced to guarantee the mentioned condition that the sum of outstanding principals of assets never exceeds the outstanding principal of the main loan. The authors also show that in the case where all assets share the same financial characteristics (amortization rule, internal interest rate and term) all but one constraint turn out to be redundant and hence the model reduces to (KP) for this special setting.

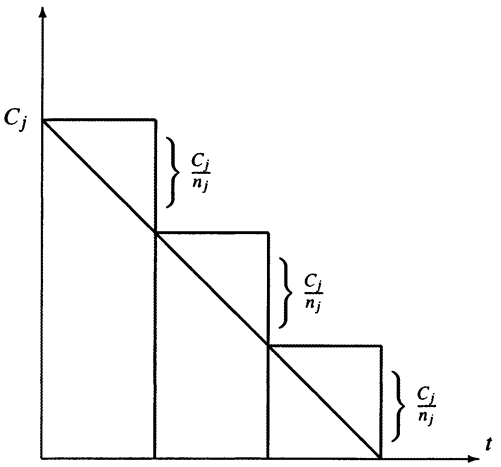


Fig. 15.3. Italian amortization: Outstanding principle of an asset j with initial outstanding principal C_j and $n_j = 3$ installments.

In the so-called *Italian amortization* the principal installments are identical over time. Hence, the outstanding principal of each asset has a very simple step-wise decreasing shape as illustrated in Figure 15.3, and does not depend on the rate of amortization. Mansini and Pferschy [317] considered a model to select financial assets *at different dates* which allows for more flexibility and better financial performance. Motivated by a real-world problem and also the fact that it is more tractable they considered *Italian* instead of French amortization. The resulting optimization problem and some approximation algorithms will be the subject of the following two sections.

15.5.2 Formal Problem Definition

Given a set of assets S with $|S| = m$ let us define as C_j the initial value of the outstanding principal of asset j (i.e. the purchasing price of a property such as machinery, vehicle or real estate) and as n_j the number of installments. Then the steps of the outstanding principal all have the same size as shown in Figure 15.3. Since the points $(0,C_j)$, $(1,(n_j-1)\frac{C_j}{n_j})$, $\ldots,(k,(n_j-k)\frac{C_j}{n_j})$ and $(n_j,0)$ lie on the same line, the outstanding principal can be approximated efficiently by a linear function.

For simplicity of presentation we introduce a discretized time horizon with points in time $0,1,2,\ldots$ (corresponding to months) for the reimbursement dates of assets. The reimbursement dates for the main loan are assumed to happen at certain fixed multiples of this time unit (usually every 12 or 18 months). These points in time are collected in a set $D := \{t_0 = 0, t_1, \ldots, t_T\}$ with the corresponding index set $\mathcal{T} = \{0,1,\ldots,T\}$.

According to the real-world situation assets should be selected at each point in time $t_i, i \in \mathcal{T}$, in such a way that the sum of their outstanding principals is as close as possible to the outstanding principal of the main loan without exceeding it. The resulting optimization problem can be formulated as follows.

$$x_{ij} = \begin{cases} 1 & \text{if asset } j \text{ is included in the portfolio at time } t_i, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{(ABS)} \quad v := \max \sum_{i=0}^{T} \sum_{j=1}^{m} p_{ij} x_{ij} \tag{15.23}$$

$$\sum_{i=0}^{T} x_{ij} \leq 1, \qquad \forall\, j \in S, \tag{15.24}$$

$$\sum_{j \in S, n_j \geq t_i} C_{ij} \sum_{k=0}^{i} x_{kj} \leq C_{i0}, \qquad \forall\, i \in \mathcal{T}, \tag{15.25}$$

$$x_{ij} \in \{0,1\}, \qquad i \in \mathcal{T},\ j \in S, \tag{15.26}$$

where C_{i0} is the outstanding principal of the main loan at time t_i with $C_{00} = C$, and n is the corresponding number of installments. The coefficients for the assets with $n_j > t_i$ are given by

$$p_{ij} := \sum_{s=0}^{n_j - t_i} \frac{C_j}{n_j}(n_j - t_i - s) = \frac{C_j}{2n_j}(n_j - t_i)(n_j - t_i + 1),$$
$$C_{ij} := \frac{C_j}{n_j}(n_j - t_i),$$

otherwise they are equal to zero. For $n_j > t_i$, C_{ij} represents the residual outstanding principal of asset j, $j \in S$, at time t_i while the value p_{ij} represents the sum of its outstanding principals from time t_i until its deadline n_j.

Exploiting the particular form of the outstanding principal computed with Italian amortization, it was shown by Mansini and Pferschy [317] that the problem of selecting the assets only at the beginning of the (ABS) process can be formulated as a simple knapsack problem (KP), where the single constraint corresponds to the asset selection at time 0.

$$\max \sum_{j \in S} p_j x_j$$
$$\sum_{j \in S} C_j x_j \leq C,$$
$$x_j \in \{0,1\}, \qquad j \in S,$$

with $p_j = \frac{C_j}{2}(n_j + 1)$.

15.5.3 Approximation Algorithms

Solving (ABS) to optimality requires in general the solution of a special version of a $(T+1)$-dimensional multiple-choice knapsack problem (d-MCKP). The difficulty of this task was pointed out in Section 9.8 (see also Section 9.3). Hence, we will proceed by describing approximation algorithms for (ABS). Proofs and more detailed elaborations are omitted and can be found in the original paper by Mansini and Pferschy [317].

In analogy to standard knapsack terminology we will define the efficiency of an asset j at time t_i by $\frac{p_{ij}}{C_{ij}} = \frac{1}{2}(n_j - t_i + 1)$, which is increasing in n_j for all i. In the remainder of this section we will assume that the assets are sorted in decreasing order of duration n_j, i.e. $n_1 \geq \ldots \geq n_m$.

At first the LP-relaxation of (ABS) where (15.26) is replaced by $x_{ij} \in [0,1]$, $i \in \mathcal{T}$, $j \in S$, will be considered. The corresponding optimal solution vector x_{ij}^{LP} has

a surprisingly simple structure and can be computed by a straightforward greedy-type algorithm ABS-LP described in Figure 15.4. The resulting block structure is illustrated by Figure 15.5, where $\lambda_0, \lambda_1, \ldots$ denote the fractional part of an asset chosen such that (15.25) is fulfilled with equality. By a simple exchange argument it was shown in [317]:

Lemma 15.5.1 *The solution x_{ij}^{LP} generated by algorithm* ABS-LP *is an optimal solution for the LP-relaxation of (ABS).* □

Algorithm ABS-LP:

for $i = 0$ to T
  for $j = 1$ to m
    choose $x_{ij}^{LP} \in [0,1]$ as large as possible
    such that (15.24) and (15.25) are fulfilled

Fig. 15.4. Greedy algorithm for the LP-relaxation of (ABS).

x_{0j}^{LP}	1	...	1	λ_0	0	...	0	0	0	...	0	0	...
x_{1j}^{LP}	0	...	0	$1-\lambda_0$	1	...	1	λ_1	0	...	0	0	...
x_{2j}^{LP}	0	...	0	0	0	...	0	$1-\lambda_1$	1	...	1	λ_2	...

Fig. 15.5. Typical structure of the solution x_{ij}^{LP} for t_0, t_1, t_2 generated by ABS-LP.

Following the structure of the optimal solution vector x^{LP}, a feasible solution can be computed by a greedy-type algorithm. It can be shown that such an algorithm may perform arbitrarily bad. Hence, a modified greedy algorithm ABS-Ext-Greedy was introduced which is slightly more complicated than Ext-Greedy for (KP) because the first fractional item (asset) has to be taken special care of. To avoid leaving out a very large "split item" s, i.e. the first asset causing a violation of constraint (15.25) at time t_0, we compute the time t_s as the earliest time where the split item s can be included in the portfolio together with the assets of longer duration 1 to $s-1$.

Then it will be determined by a comparison of profits whether this "split item" should be selected already at time t_0 and all assets 1 to $s-1$ later at time t_s or the other way round, namely selecting asset s at time t_s and the assets 1 to $s-1$ at the closing time t_0. After this decision a simplified greedy algorithm deals with the remaining assets (see Figure 15.6).

The running time of the main computation of ABS-Ext-Greedy is $O(m+T)$, since both assets and time periods are considered in increasing order without going back. Its performance guarantee was proved in Mansini and Pferschy [317].

Algorithm ABS-Ext-Greedy:

compute asset $s := \min\{k \mid \sum_{j=1}^{k} C_{0j} > C_{00}\}$

compute time $t_r := \min\{t \mid \sum_{j=1}^{s} C_{rj} \leq C_{r0}, r \in \mathcal{T}\}$.

if $\sum_{j=1}^{s-1}(p_{0j} - p_{rj}) \geq p_{0s} - p_{rs}$ then

  select assets 1 up to $s-1$ at time t_0 and asset s at time t_r

else

  select asset s at time t_0 and assets 1 up to $s-1$ at time t_r

$j = s+1$

for $i = r+1$ to T

  while (15.25) permits

    select asset j at time t_i and increment j

  increment j *the split item is omitted in every period*

Fig. 15.6. Modified greedy algorithm for (ABS).

Theorem 15.5.2 *Algorithm* ABS-Ext-Greedy *has a relative performance guarantee of* $\frac{1}{2}$ *for (ABS) and this bound is tight.* □

Another obvious approach to (ABS), which may perform better than the adaptation of the greedy approach and the LP-relaxation, is the solution of $T+1$ different knapsack problems. This means that we first consider only the assets at time t_0 and solve the resulting subproblem, which is an instance of (KP). Then we optimize separately the portfolio at time t_1 and so on.

Although it may seem strange to propose the solution of an $\mathcal{NP}$–hard problem for a heuristic, it was pointed out in Chapter 5 that several promising strategies are available for the optimal solution of (KP).

The resulting heuristic ABS-Knap is performed in $T+1$ steps by solving iteratively for $i = 0, \ldots, T$ the following instance $(\text{KP})_i$ with optimal variable values x_{ij}^*.

$$(\text{KP})_i \qquad v_i^K = \max \sum_{j=1}^{m} p_{ij} x_{ij}$$

$$\sum_{k=0}^{i-1} x_{kj}^* + x_{ij} \leq 1, \qquad \forall\, j \in S, \tag{15.27}$$

$$\sum_{j \in S, n_j \geq t_i} C_{ij} \left(\sum_{k=0}^{i-1} x_{kj}^* + x_{ij} \right) \leq C_{i0}, \tag{15.28}$$

$$x_{ij} \in \{0,1\}, \qquad j = 1, \ldots, m.$$

Note that $(\text{KP})_i$ is indeed a standard knapsack problem since constraint (15.27) can be handled by eliminating from consideration any asset j where $\sum_{k=0}^{i-1} x_{kj}^* = 1$. The

overall solution value of this heuristic is given by $v^K = \sum_{i=0}^{T} v_i^K$. It was shown by Mansini and Pferschy [317] with some technical effort that the worst-case performance of ABS-Knap is the same as for ABS-Ext-Greedy.

Theorem 15.5.3 *Algorithm* ABS-Knap *has a relative performance guarantee of* $\frac{1}{2}$ *for (ABS) and this bound is tight.* □

Computational experiments indicate that the assets selected at time t_0 contribute by far the largest part of the objective function. To exploit this behaviour algorithmically, it should be noted that on one hand the selection at time t_0 is crucial and should be done as good as possible, whereas on the other hand the remaining points in time will not induce too critical deviations from the optimum and it is not worth to spend large amounts of computation time on them.

Realizing this properties, a mixture of ABS-Knap and ABS-Ext-Greedy was proposed in [317]. The resulting algorithm ABS-Comb solves a knapsack problem for t_0, i.e. the above instance KP_0, and applies the corresponding part of ABS-Ext-Greedy for the remaining points in time t_1 until t_T. The details of this method are obvious.

Based on a real-world situation of an Italian leasing bank, Mansini and Pferschy [317] performed extensive computational experiments with different data sets to compare ABS-Ext-Greedy, ABS-Knap, ABS-Comb and and an optimal solution computed by the commercial solver CPLEX (see `www.ilog.com`). The occurring instances of (KP) were solved by algorithm Minknap described in Section 5.4.2.

The results of their experiments can be summarized as follows: Solving (ABS) to optimality even with the best available standard software tool can be done with full justification in reasonable time only for problems with up to 100 assets. The presented approximation algorithms may have a disappointing worst-case ratio of $1/2$ but perform very well for the tested instances since their worst-case behaviour occurs only for instances with very large assets which can be ruled out in practical applications.

The best solution values were derived by ABS-Knap followed closely by ABS-Comb and ABS-Ext-Greedy. It follows immediately from their definitions that the running times of the algorithms decrease in this order. As a general rule, it is suggested to use ABS-Comb which offers the best trade-off between solution quality and running time. Detailed remarks and further observations about the particular behaviour of the algorithms can be found in [317].

15.6 Knapsack Cryptosystems

Public key cryptosystems have become widely known after Rivest, Shamir and Adleman [408] discovered in 1976 their celebrated RSA public key cryptosys-

tem which is based on factoring large integers. In the same year Merkle and Hellman [345] proposed the first so-called *knapsack cryptosystem*. This class of cryptosystems has been an early alternative to the RSA system and belongs to one of the few classes of public key cryptosystems. The name knapsack cryptosystem is misleading since all these cryptosystems are in fact based on the subset sum problem not on the knapsack problem. Moreover, the considered problem is not (SSP) but related to its decision version, namely SSP-DECISION, as introduced in the beginning of Chapter 4. We call this subset sum problem *exact subset sum problem*, briefly (ESSP). While SSP-DECISION only asks whether there is a solution with total weight c, (ESSP) looks, in case the answer is yes, also for the corresponding solution vector. Thus, (ESSP) can be described as follows: Given a set of positive integers $\{w_1,\ldots,w_n\}$ and a specific value c, find a subset of $\{w_1,\ldots,w_n\}$ with total sum equal to c (if such a subset exists), or equivalently find a binary solution vector $x=(x_1,\ldots,x_n)$ such that $\sum_{j=1}^{n} w_j x_j = c$. Note that (ESSP) and SSP-DECISION have different complexity for cryptosystems since we know only for weight c that the answer is yes, but there is no oracle which answers the question for the existence of solutions with weights smaller than c.

A comprehensive introduction into public key cryptography can be found in the book by Salomaa [416]. The paper by Brickell and Odlyzko [54] on breaking knapsack cryptosystems has inspired parts of this section.

15.6.1 The Merkle-Hellman Cryptosystem

A knapsack cryptosystem works in principle as follows: Assume Bob wants to send Alice a secret message called LOVE. In the binary system the word LOVE can be e.g. written as a binary vector x with

$$\text{LOVE} = x = (0,1,1,0,0;0,1,1,1,1;1,0,1,1,0;0,0,1,0,1),$$

where five digits represent one letter of the Latin alphabet. Alice publishes the vector $w=(w_1,\ldots,w_n)$ as a so-called *public key*. A message $x=(x_1,\ldots,x_n)$ of Bob is encoded as

$$c=\sum_{j=1}^{n} w_j x_j,$$

and transmitted to the receiver Alice, i.e. she will get c with

$$c = w_2 + w_3 + w_7 + w_8 + w_9 + w_{10} + w_{11} + w_{13} + w_{14} + w_{18} + w_{20}.$$

An unauthorized person must solve (ESSP) in order to reconstruct the message from Bob out of c which becomes intractable for sufficiently large values of n. But how can Alice read the message from Bob? She constructs a vector $v=(v_1,\ldots,v_n)$ for which the corresponding subset sum problem is polynomially solvable for every value of c. This vector is transformed into the vector w. The *private key* of Alice

is a method to reconstruct the message from c by solving the "easy" subset sum problem with weights $v_1,\ldots,v_n$. Summarizing, a classical knapsack cryptosystem can be described as follows:

1. Look for positive integers $v_1,\ldots,v_n$ for which (ESSP) is "easy" (polynomially solvable).
2. Transform the integers $v_1,\ldots,v_n$ into numbers $w_1,\ldots,w_n$ for which (ESSP) is supposed to be "hard". The numbers $w_1,\ldots,w_n$ form the public key.
3. Find a method to reconstruct the message $x=(x_1,\ldots,x_n)$ from c by solving the "easy" subset sum problem with weights $v_1,\ldots,v_n$. The method is kept secret as the private key.

Merkle and Hellman [345] use numbers $v_1,\ldots,v_n$ which form a *superincreasing sequence*, i.e.

$$v_j > \sum_{i=1}^{j-1} v_i, \quad 2 \le j \le n. \tag{15.29}$$

The receiver uses integers m and t with

$$m > \sum_{j=1}^{n} v_j \tag{15.30}$$

and t, m relatively prime, i.e. $\gcd(t,m)=1$. Then the public key w will be calculated by

$$\tilde{v}_j \equiv v_j t \pmod{m}, \quad j=1,\ldots,n, \tag{15.31}$$

and setting

$$w_j = \tilde{v}_{\pi(j)}, \quad j=1,\ldots n, \tag{15.32}$$

where π is a permutation of $(1,\ldots,n)$ which is introduced to achieve further security. The values m, t and the permutation π are the elements of the private key. Let t^{-1} be the multiplicative inverse of t modulo m, i.e. t^{-1} is the unique element of $\{1,\ldots,m-1\}$ for which there is an integer k with

$$t\,t^{-1} = mk+1.$$

Note that t^{-1} is well-defined since t and m are relatively prime. After receiving a message $(x_1,\ldots,x_n)$ encoded as $c=\sum_{j=1}^{n} w_j x_j$ compute

$$\tilde{c} \equiv c t^{-1} \pmod{m}. \tag{15.33}$$

By (15.31) and (15.32) the value $\tilde{c}$ can be expressed as

$$\begin{aligned}\tilde{c} &\equiv t^{-1}\sum_{j=1}^{n} w_j x_j \pmod{m} \\ &\equiv t^{-1}\sum_{j=1}^{n} \tilde{v}_{\pi(j)} x_j \pmod{m} \\ &\equiv \sum_{j=1}^{n} v_{\pi(j)} x_j \pmod{m}.\end{aligned}$$

Because of (15.30) we have even

$$\tilde{c} = \sum_{j=1}^{n} v_{\pi(j)} x_j. \tag{15.34}$$

Since the values $v_1, \ldots, v_n$ form a superincreasing sequence, the original message x can now easily be decrypted.

Let us mention that due to the choice of a superincreasing sequence, there is always a unique solution of the corresponding exact subset sum problem with capacity c, i.e. the message x is always unique for a given c.

The cryptosystem described above by Merkle and Hellman is a *singly-iterated cryptosystem*. In their paper they proposed also a *multiply-iterated cryptosystem* in which the integers corresponding to an easy subset sum problem are hidden by several modular multiplications.

15.6.2 Breaking the Merkle-Hellman Cryptosystem

After publishing their cryptosystem in 1976 Merkle and Hellman were quite sure that it is secure and Merkle offered 100 $ for breaking it. But in 1982 Shamir [430] was able to find a polynomial time algorithm breaking the singly-iterated cryptosystem. Still Merkle was sure that at least the multiply-iterated cryptosystem was secure and he offered 1000 $ for breaking it. In 1984 Brickell [52] succeeded in breaking also this system.

We will now give a short description of the method of Shamir to break the singly-iterated cryptosystem. Let π^{-1} denote the inverse permutation of π. From (15.31) and (15.32) we deduce with $\tilde{w}_j := w_{\pi^{-1}(j)}$

$$\tilde{w}_j \equiv v_j t \pmod{m}.$$

Thus, there are integers $k_1, \ldots, k_n$ with

$$v_j = \tilde{w}_j t^{-1} - k_j m, \quad j = 1, \ldots, n, \tag{15.35}$$

or equivalently,

$$\frac{t^{-1}}{m} - \frac{k_j}{\tilde{w}_j} = \frac{v_j}{m\tilde{w}_j}. \tag{15.36}$$

By multiplying (15.35) for some $j > 1$ with $\tilde{w}_1$ and multiplying (15.35) again for $j = 1$ with $\tilde{w}_j$ $(j > 1)$, we get after subtracting both equations

$$v_j\tilde{w}_1 - v_1\tilde{w}_j = mk_1\tilde{w}_j - mk_j\tilde{w}_1,$$

which results in

$$k_j\tilde{w}_1 - k_1\tilde{w}_j = \frac{v_1\tilde{w}_j - v_j\tilde{w}_1}{m}, \quad j = 2,\ldots,n. \tag{15.37}$$

Since the numbers v_j are superincreasing, we can show

$$v_j < m2^{j-n}, \quad j = 1,\ldots,n-1. \tag{15.38}$$

Otherwise, there is an $i < n$ with $v_i \geq m2^{i-n}$ and we get with (15.29) that $v_{i+j} > 2^{j-1}m2^{i-n}$ for $k = 1,\ldots,n-i$. The sum of these values yields

$$\sum_{j=i}^{n} v_j > m2^{i-n} + m2^{i-n}\sum_{j=0}^{n-i-1} 2^j =$$

$$= m2^{i-n}\left(1 + \frac{2^{n-i}-1}{2-1}\right) = m,$$

a contradiction to (15.30).

With (15.38) and since $\tilde{w}_j \leq m$, it is possible to bound the absolute value of(15.37) for $j < n$ as follows

$$|k_j\tilde{w}_1 - k_1\tilde{w}_j| \leq m2^{j-n}, \quad j = 1,\ldots,n-1. \tag{15.39}$$

Analogously, we get for (15.36)

$$\left|\frac{t^{-1}}{m} - \frac{k_j}{\tilde{w}_j}\right| \leq 2^{j-n}. \tag{15.40}$$

We show now how it is possible to break the code with the knowledge of a few values $\tilde{w}_j$ for small j. Recall that only the values w_j are public. First of all, we "guess" from $w_1,\ldots,w_n$ e.g. the values $\tilde{w}_1,\ldots,\tilde{w}_4$ which correspond to the smallest values $v_1,\ldots v_4$ of the superincreasing sequence $v_1,\ldots,v_n$. This can be done by checking all possibilities for quadruples $(\tilde{w}_1,\tilde{w}_2,\tilde{w}_3,\tilde{w}_4)$ in $O(n^4)$ time.

By (15.39) the difference $|k_j\tilde{w}_1 - k_1\tilde{w}_j|$ must be very small, especially for $j \leq 4$. In other words, $k_1\tilde{w}_j$ is close to $0 \pmod{\tilde{w}_1}$. Taking into account that the values $k_1\tilde{w}_j \pmod{\tilde{w}_1}$ should normally be uniformly distributed in $0,\ldots,\tilde{w}_1 - 1$, the inequalities (15.39) suffice with very high probability to determine the values $k_1,\ldots,k_4$ uniquely.

If the values $k_1,\ldots,k_4$ are found, it is easy to find an approximation to $\frac{t^{-1}}{m}$ from (15.40) and to construct values $\hat{t}^{-1}$, $\hat{m}$ with $\frac{\hat{t}^{-1}}{\hat{m}}$ close to $\frac{t^{-1}}{m}$ such that the values

$$\hat{v}_j \equiv w_j \hat{t}^{-1} \pmod{\hat{m}}, \quad j = 1,\ldots,n \tag{15.41}$$

form a superincreasing sequence after sorting them in increasing order. Note that any superincreasing sequence which fulfills (15.41) suffices to reconstruct the message. Thus, no explicit knowledge about the real values of t and m is necessary.

It remains to present an algorithm for determining the values k_j from (15.39). Shamir recommended to solve the integer linear program

$$\begin{aligned} &\text{minimize } \delta \\ &\text{subject to } k_j\tilde{w}_1 - k_1\tilde{w}_j \leq \delta, \\ &\qquad\qquad k_j\tilde{w}_1 - k_1\tilde{w}_j \geq -\delta \\ &\qquad\qquad k_j \text{ integer}, \quad j = 1,\ldots,4, \end{aligned}$$

by employing the famous theorem of Lenstra [298] that an integer linear program with a fixed number of variables can be solved in polynomial time. Unfortunately, the degree of the polynomial in Lenstra's algorithm is extremely high. So, his approach is of limited practical use. A short time later Adleman [4] treated the Merkle-Hellman cryptosystem as a lattice problem rather than a linear programming problem. He discovered that the so-called *Lovász lattice reduction algorithm* by Lenstra, Lenstra and Lovász [297] was an efficient tool to replace Lenstra's algorithm for breaking the Merkle-Hellman cryptosystem.

15.6.3 Further Results on Knapsack Cryptosystems

Despite the breakings of the cryptosystems by Merkle and Hellman researchers continued looking for improved knapsack cryptosystems since such systems are very easy to implement and their running times are much faster than the RSA cryptosystem. Nearly all these cryptosytems have been broken. Many of these attacks are *lattice based* attacks based on the Lovász lattice reduction algorithm. As mentioned above, they were introduced by Adleman [4] for breaking the singly-iterated cryptosystem by Merkle and Hellman. This technique was also studied by Brickell, Lagarias and Odlyzko [53] and Lagarias [292]. Later Brickell [52] succeeded in breaking the multiply-iterated cryptosystem using this method. Further improvements of the Lovász lattice reduction algorithm have been derived by Schnorr [426, 427].

The *density of a subset sum problem* is defined to be

$$\frac{n}{\log_2(w_{\max})}. \tag{15.42}$$

Low density attacks, i.e. methods to break knapsack cryptosystems with low density, were introduced by Brickell [51] and Lagarias and Odlyzko [293].

One of the strongest knapsack cryptosystems was the cryptosystem by Chor and Rivest [81] developed in 1988, used high density knapsacks. There have been several unsuccessful attacks on that system, but finally it was broken in 1998 by Vaudenay [472] with algebraic techniques. The leading candidate among knapsack-based and lattice-based cryptosystems to survive is the NTRU cryptosystem [234] proposed by Hoffstein, Pipher and Silverman (see also `http://www.ntru.com` for further details).

15.7 Combinatorial Auctions

Auctions have been a topic of research of market mechanism in the economic literature. In 1961 Vickrey [473] was the first to analyze auctions from a game-theoretic point of view. In the last few years there has been a great rise of interest in various types of auctions due to the rapid increase of the importance of electronic commerce applications.

In many cases the price which a bidder offers for some good may depend on the other goods he buys. Thus, it seems advantageous that prices are offered not for single goods but for sets of goods (items) to be sold. The corresponding type of auction is called *combinatorial auction.*

A combinatorial auction is of specific interest if some goods are considered to be complementary for the bidder, i.e. the value of a set of items is higher than the sum of the values of the single items. As an example consider a technical equipment consisting of several parts which has a higher value if bought as a whole bundle. Another example is London transport. The company has sold in auction about 800 bus routes per year since 1995 by a combinatorial auction.

There are several ways of *winner determination*, i.e. to determine those sets of bids which are accepted by the auctioneer. One possible way is defined as the *combinatorial auction problem*: An auctioneer gets price offers for sets of items and his goal is to allocate the items to the bidders such that the turnover, reflecting his personal revenue, is maximized.

Formally, in the *combinatorial auction problem* we have given a set $N = \{1,\ldots,n\}$ of n bids and a set $D = \{1,\ldots,d\}$ of d goods. A bid j is defined as a pair (A_j, p_j) where A_j denotes a subset of the set of goods D and p_j is the price offer for A_j. The auctioneer has to select a set of disjoint sets A_j with maximal total price. Set $a_{ij} := 1$ if good i is contained in bid j, i.e. $i \in A_j$, otherwise $a_{ij} := 0$. Let the binary variable x_j determine whether bid j is selected by the auctioneer or not. Then, the integer programming formulation of the combinatorial auction problem reads as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq 1, \quad i = 1,\ldots,d, \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned} \tag{15.43}$$

This combinatorial auction model is also called a *single-unit combinatorial auction*, because there is exactly one copy of each item available. Since the problem is in principle equivalent to a weighted *set packing problem*, it is strongly $\mathcal{NP}$-hard (see Garey and Johnson [164, SP3]).

Combinatorial auctions have been investigated for the first time in 1982 by Rassenti, Smith and Bulfin [404], in the context of selling time-slots at airports in order to permit airlines to bid simultaneously for take-off and landing time-slots. The research on combinatorial auctions splits into papers on polynomially solvable subcases, e.g. problems with infinitely divisible goods which are optimally solvable using linear programming techniques, and into heuristics and branch-and-bound algorithms for the general problem. For a recent survey on the literature on combinatorial auctions we refer to de Vries and Vohra [103].

15.7.1 Multi-Unit Combinatorial Auctions and Multi-Dimensional Knapsacks

In a single-unit combinatorial auction we have only one unit of each item. An obvious generalization are the so-called *multi-unit combinatorial auctions*. These are auctions in which there is a fixed number of identical units of each item. Formally, in a *multi-unit combinatorial auction problem* we have given a set $N = \{1,\ldots,n\}$ of n bids and a set $D = \{1,\ldots,d\}$ of d goods. A bid j can be defined as a pair (A_j, p_j) where p_j is the price offer of bid j and A_j is a vector with $A_j := (a_{1j},\ldots,a_{dj})$. Here, a_{ij} denotes the number of copies of item i requested by bid j. Let c_i denote the number of available units of good i, $i = 1,\ldots,d$. The auctioneer has to select a set of bids with maximal total price value such that the total number of requested copies of each item does not exceed the total number of available copies. Let the binary variable x_j determine again whether bid j is selected or not. Then, the integer programming formulation of the multi-unit combinatorial auction problem reads as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq c_i, \quad i = 1,\ldots,d, \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,n. \end{aligned} \tag{15.44}$$

Thus, a multi-unit combinatorial auction problem is exactly a d-dimensional knapsack problem (d-KP) where each good reflects one constraint and selecting a bid corresponds to putting it into the knapsack.

Note, that this formulation allows also to model the situation where a participant of the auction submits several bids and it is still guaranteed that not more than one of his bids is selected by the auctioneer. This is done by introducing *dummy goods* with price zero: If bids i and j shall be mutually exclusive, a single-unit item δ is added to both bids. The fact that only one unit of δ is available, ensures that bids i and j are not selected simultaneously.

Multi-unit combinatorial auctions were introduced by Leyton-Brown, Shoham and Tennenholtz [300] in 2000 but they did not see the relation to (d-KP). For determining the optimal set of bids the authors propose a branch-and-bound algorithm, called CAMUS. Every time a bid is selected, CAMUS computes an upper bound on the total price offer by the remaining unallocated goods. Some experimental results with random distributions are given.

Holte [236] detected in 2001 that multi-unit combinatorial auction problems are identical to multi-dimensional knapsack problems. Holte proposes several greedy-type heuristics which he called *hill-climbing algorithms*and which can be considered as variants of the primal greedy heuristic of Section 9.5.1. Bids are selected iteratively and every time among the remaining feasible bids the one with the highest "score" is taken. Three different ways of defining a score of a bid are given:

- the price of the bid p_j,
- the price of the bid divided by its "size" $p_j/\sqrt{\sum_{i=1}^{d} f_{ij}^2}$, where f_{ij} denotes of the remaining quantity of good i that bid j requires,
- the price of the bid divided by its *knockout cost*, i.e. the sum of the prices of the available bids that have to be rejected in the case this bid is selected.

Holte also considered randomized hill-climbing where a bid is chosen randomly and the probability of choosing a bid is proportional to its score. Test problems were generated using the so-called CATS suite of problem generators for modelling particular realistic scenarios in which combinatorial auctions may arise [299]. It turned out that there was not much difference between the various hill-climbers, but a randomized hill-climbing procedure usually outperformed the corresponding deterministic version.

Gonen and Lehmann [190] apply linear programming relaxations to derive upper bounds for the value of the optimal solution (see also [189]). Numerical experiments on "realistic" data distributions show that the gap between lower bounds from greedy heuristics and the linear programming upper bounds are relatively small and thus in a branch-and-bound procedure pruning is extensive.

As an example for a recent paper we refer to Archer, Papadimitriou, Talwar and Tardos [13] who use randomized rounding-based approximation algorithms for de-

signing truthful multi-unit combinatorial auctions. An auction is called *truthful* or *incentive compatible* if the best strategy for each bidder is to reveal his true valuation for his bid, independently of the bids or valuations of the other bidders.

15.7.2 A Multi-Unit Combinatorial Auction Problem with Decreasing Costs per Unit

In this section we give an example for a special multi-unit combinatorial auction problem which can be approximated efficiently by a knapsack-type *FPTAS*. This single-good multi-unit combinatorial auction problem was presented in 2003 by Kothari, Parkes and Suri [288].

Only one good is given of which c units are available. The n bidders are willing to buy any amount of the good between some personal lower and upper bound on the number of requested items. So, in this case a bid is a decreasing price function $p = p(q)$ which gives the price offer for q units of the good. More specifically, assume that $u_j^1 < \ldots < u_j^{n_j}$, then the price function p_j of bid j is $p_j(q) = qp_j^i$, if $u_j^i \leq q < u_j^{i+1}$ for $i = 1,\ldots,n_j - 1$. Additionally, the price offer is 0 for quantities less than the minimum quantity u_j^1 as well as for quantities larger than the maximum offer $u_j^{n_j}$. Thus, p_j can be completely described by the list

$$L_j := ((u_j^1,p_j^1),(u_j^2,p_j^2),\ldots,(u_j^{n_j-1},p_j^{n_j-1}),(u_j^{n_j},0)). \tag{15.45}$$

Moreover, the prices p_j^i shall be sorted in decreasing order, i.e.

$$p_j^1 > p_j^2 > \ldots > p_j^{n_j-1}. \tag{15.46}$$

Consequently, the average unit price function $p_j(q)/q$ is a marginal-decreasing piecewise constant curve with breakpoints $u_j^1,\ldots,u_j^{n_j}$.

Kothari, Parkes and Suri defined a *generalized knapsack problem*, which will be denoted as (GKP). Problem (GKP) is equivalent to their multi-unit combinatorial auction problem.

An instance of (GKP) consists of an upper bound c and n lists L_j as defined in (15.45) with the values p_j^i decreasing and the values u_j^i increasing, i.e. with $p_j^1 > \ldots > p_j^{n_j-1}$ and $u_j^1 < \ldots < u_j^{n_j}$ for $j = 1,\ldots,n$ for positive integers u_j^i, p_j^i and c.

(GKP) looks for integers x_j^i such that the revenue

$$\sum_{j=1}^{n}\sum_{i=1}^{n_j} p_j^i x_j^i$$

is maximized under the following constraints:

1. At most one x_j^i is non-zero for $j = 1,\ldots,n$.
2. $x_j^i \neq 0$ implies $x_j^i \in [u_j^i, u_j^{i+1}[$ for $j = 1,\ldots,n$ (setting $u_j^{n+1} := \infty$).
3. $\sum_{j=1}^{n} \sum_{i=1}^{n_j} x_j^i \leq c$.

Variable x_j^i denotes the amount of items sold to bidder j. If $x_j^i > 0$, constraint 1 means that $u_j^i \leq x_j^i < u_j^{i+1}$. Constraint 2 ensures that x_j^i is unique for every j. The fact that not more than c units of the good are available is represented by the capacity constraint 3.

It is obvious that (GKP) is similar to the multiple-choice knapsack problem (MCKP) investigated in Section 11. In fact, (GKP) can be converted into a huge instance of (MCKP) by creating one class of items for each list and each class contains different items corresponding to the different amounts of possible quantities. But nevertheless, a direct *FPTAS* for (GKP) can be found by scaling the price range. Under the assumption that the values n_j are constants, Kothari, Parkes and Suri presented a 2-approximation algorithm for (GKP) and an *FPTAS* with running time in $O(n^3/\varepsilon)$.

A. Introduction to $\mathcal{NP}$-Completeness of Knapsack Problems

The reader may have noticed that for all the considered variants of the knapsack problem, no polynomial time algorithm have been presented which solves the problem to optimality. Indeed all the algorithms described are based on some kind of search and prune methods, which in the worst case may take exponential time. It would be a satisfying result if we somehow could prove it is not possible to find an algorithm which runs in polynomial time, somehow having evidence that the presented methods are "as good as we can do". However, no proof has been found showing that the considered variants of the knapsack problem cannot be solved to optimality in polynomial time.

The theory of $\mathcal{NP}$-completeness gives us a framework for showing that it is very doubtful that a polynomial algorithm exists. Indeed, if we could find a polynomial algorithm for solving e.g. the subset sum problem, then we would also be able to solve numerous famous optimization problems like the *traveling salesman problem*, *general integer programming*, and we would even be able to efficiently find *mathematical proofs* of theorems, as stated in Cook [91]. A very comprehensive guide to the theory of $\mathcal{NP}$-completeness is found in the seminal book by Garey and Johnson [164]. Simplified introductions can be found in nearly any text book on combinatorial optimization or algorithms, e.g. in Cormen et al. [92], or Papadimitriou and Steiglitz [367]. A compendium on $\mathcal{NP}$-optimization problems can be found in Crescenzi and Kann [94].

A.1 Definitions

The following discussion is based on the assumption that problems which are solvable in polynomial time, somehow are *tractable*. As introduced in Section 1.5, we will use the big-Oh notation to describe the asymptotic running time of an algorithm. Although a running time of $O(n^{100})$ may not seem very attractive, polynomial running times have some nice properties which support this assumption. Moore's law [348] argues that the speed of computers is doubled every 18 months. For a polynomially solvable problem, each time the speed of the computer gets doubled, we will be able to solve problems which are larger by a multiplicative factor, while we only

H. Kellerer et al., *Knapsack Problems*
© Springer-Verlag Berlin Heidelberg 2004

get an additive increase for problems demanding an exponential number of iterations. If for instance a problem is solvable in $O(n^3)$ then each time the speed of the computer is doubled, we may solve problems which are $\sqrt[3]{2}$ larger using the same computational time. If a problem is solvable in time $O(2^n)$ then each time the speed of the computer is doubled we may solve problems which are only a single decision variable larger within the same computational time.

We will restrict our discussion to *decision problems*, i.e. problems which may be answered by a "yes" or a "no". An optimization problem is easily transformed to a decision problem by comparing the solution value with a threshold value. For instance the knapsack problem in decision form asks whether a solution to the knapsack problem with objective value larger than t exits. Hence, the optimization and decision problems may be stated as:

$$\text{KP-OPTIMIZATION}(p,w,c) = \left\{ \begin{array}{l} \max \sum_{j=1}^{n} p_j x_j \\ \text{s.t.} \sum_{j=1}^{n} w_j x_j \leq c, \\ \quad x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\} \tag{A.1}$$

$$\text{KP-DECISION}(p,w,c,t) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} p_j x_j \geq t, \\ \sum_{j=1}^{n} w_j x_j \leq c, \\ x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\} \tag{A.2}$$

If we are able to solve the decision problem efficiently we are generally also able to solve the optimization problem efficiently. As an example we may solve KP-OPTIMIZATION by using *binary search* for the optimal solution value by a number of calls to KP-DECISION. Knowing that the optimal solution must be between $a := 0$ and $b := \sum_{j=1}^{n} p_j$, we make a call to KP-DECISION using the threshold value $t := \frac{a+b}{2}$. If we get the answer "yes" we set $a := t$ otherwise we set $b := t - 1$ and repeat the process. When $a = b$ we have the optimal solution value. The number of calls to KP-DECISION is bounded by $\log(\sum_{j=1}^{n} p_j)$ which is polynomial in the length of the input. If KP-DECISION could be solved in polynomial time we could also solve KP-OPTIMIZATION in polynomial time.

While KP-DECISION is a general problem, an *instance* of KP-DECISION is a concrete dataset $I = \{p_1,\ldots,p_n,w_1,\ldots,w_n,c\}$. This could e.g. be the following dataset

j	1	2	3	4
p_j	8	5	2	3
w_j	5	2	4	8

$c = 10,\ t = 9$ (A.3)

It is easy to see that the instance is a "yes"-instance since choosing item 1 and 3 gives a feasible solution with profit sum of 10, exceeding the threshold value t.

Since we measure the time and space complexity as a function of the *input size* $L(I)$, one must define some standards how the input size of an instance I is measured. We will use the natural assumption that all input data is written in binary form, and

hence the input size is the number of bits needed to represent the instance. For the KP-DECISION problem it is common to assume that all weights are smaller than c and all the profits are smaller than t. Hence, we will need to use $\log_2 c$ bits to represent c and each of the profits and weights w_j. In a similar way, we note that $\log_2 t$ bits are needed to represent t and each of the profits. The total input size of instance I becomes $L(I) = (n+1)\log_2 c + (n+1)\log_2 t$ which is bounded by $O(n \log c + n \log t)$. For the example in (A.3) we note that 4 bits are needed to represent c and t hence the total input size is $10 \cdot 4 = 40$.

We say that an algorithm *accepts* a decision problem in polynomial time if there exists an algorithm which runs in time polynomial in the input size and has the following property: For every instance corresponding to the decision problem, if it is a "yes" instance, then the algorithm should print out "yes" in polynomial time. The definition does not put any restrictions on "no" instances, hence the algorithm may print out "no" in polynomial time, or it may simply be the case that it does not terminate in polynomial time for a "no" instance (or even it may not terminate at all).

An algorithm *decides* a decision problem if for every binary input string, it prints out "yes" if the string represents a "yes"-instance, and it prints out "no" for all other strings.

The class of *polynomially solvable problems*, or simply $\mathcal{P}$ *problems* is the set of decision problems for which an algorithm exists which can decide the problem in polynomial time.

Obviously, most researchers do not believe that KP-DECISION belongs to the class $\mathcal{P}$, since no algorithm has been found which can accept the problem in polynomial time. However, since no proof has been found which states that such an algorithm does not exist, we cannot for sure exclude KP-DECISION from the class $\mathcal{P}$.

Instead of focusing on problems which can be solved in polynomial time, we could look at the somehow larger class of problems where a solution can be verified in polynomial time. A *verification algorithm* reads a proposed solution, also called a *certificate*, and checks whether it is a proper solution to the given decision problem.

For KP-DECISION a natural choice of the certificate C is a list of the indices corresponding to the chosen items. For the instance (A.3), the certificate becomes $C = \{1,3\}$. A verification algorithm simply checks whether

$$\sum_{j \in C} p_j \geq t \text{ and } \sum_{j \in C} w_j \leq c. \tag{A.4}$$

If both criteria are satisfied, the algorithm returns "yes". It is obvious that the verification algorithm for KP-DECISION runs in polynomial time, hence we have the following:

Lemma A.1.1 KP-DECISION *can be verified in polynomial time.*

The class of $\mathcal{NP}$ *problems* is the set of problems for which a polynomial time verification algorithm exists. More formally we say that a problem $Q \in \mathcal{NP}$ if and only if there exists a two-input polynomial-time algorithm $A(I,C)$ and a constant k such that it satisfies:

$$\text{For all "yes" instances } I \in Q \text{ there exists a certificate } C \text{ with } L(C) = O(L(I)^k) \text{ such that } A(L,C) \text{ outputs "yes".} \tag{A.5}$$

Note that the certificate C must have a size $L(C)$ polynomial in the instance size $L(I)$. Hence, e.g. choosing C as all feasible solutions to KP-OPTIMIZATION is not a valid certificate.

We say that a decision problem Q can be *transformed* (or *reduced*) to a decision problem R in polynomial time, if there exists a polynomial time mapping f of every possible binary string I such that if the string I represents a "yes" instance in Q then also $f(I)$ represents a "yes" instance in R, and vice-versa. We will use the terminology

$$Q \leq_p R \tag{A.6}$$

to denote that Q can be reduced to R in polynomial time. The symbol $\leq_p$ can be interpreted as kind of hardness ordering, since if R is solvable in polynomial time, then Q is also solvable in polynomial time.

The class of $\mathcal{NP}$*-complete problems* $\mathcal{NPC}$ is the set of decision problems Q satisfying the following two properties:

1. $Q \in \mathcal{NP}$
2. $\forall R \in \mathcal{NP} : R \leq_p Q$

Informally speaking an $\mathcal{NP}$-complete problem should belong to the class $\mathcal{NP}$, i.e. a proposed solution can be verified in polynomial time, and the problem should be "harder" to solve than any other problem in the class $\mathcal{NP}$. If Q only satisfies property 2. we say that the problem is $\mathcal{NP}$*-hard.*

The $\mathcal{NP}$-complete problems are our main clue to the discussion whether $\mathcal{P} = \mathcal{NP}$. If we could find a polynomial algorithm for a problem $Q \in \mathcal{NPC}$ then property 2. above immediately gives us a polynomial algorithm for every problem in $R \in \mathcal{NP}$, since we simply can transform R to Q in polynomial time, and then solve Q in polynomial time. On the other hand, if we could prove that an $\mathcal{NP}$-complete problem R cannot be solved in polynomial time, then due to property 2. we can conclude that no $\mathcal{NP}$-complete problem has a polynomial time algorithm. To see the latter, one may simply assume that an $\mathcal{NP}$-complete problem Q could be solved in polynomial time, then by transforming R to Q we would also get a polynomial algorithm for R contradicting the assumption.

While property 1. in the definition of $\mathcal{NP}$-complete problems is quite easy to check, it is more challenging to show that *all* decision problems in $\mathcal{NP}$ can be reduced to Q in polynomial time. Hence, to prove property 2. for a given problem Q, a common

technique is to choose a known $\mathcal{NP}$-complete problem S and show the reduction $S \leq_p Q$. Since S is $\mathcal{NP}$-complete, we have

$$\forall R \in \mathcal{NP} : R \leq_p S \leq_p Q$$

which gives the required property 2. since $\leq_p$ is transitive.

Cook in his famous paper [90] from 1971 showed that the two decision problems FORMULA-SATISFIABILITY and 3CNF-SATISFIABILITY are $\mathcal{NP}$-complete by explicitly proving property 2. Most proofs of $\mathcal{NP}$-completeness are hence based on the reduction from these problems, although often through a chain of $\mathcal{NP}$-complete problems.

Even among the $\mathcal{NP}$-complete problems there are differences in the hardness of solution. A problem is solvable in *pseudopolynomial time*, if an algorithm exists which can decide the problem, and the algorithm runs in time polynomial in the number of input coefficients n and the magnitude of the coefficients. KP-DECISION is such a problem, since we may decide the problem in time $O(nc)$, using dynamic programming (e.g. DP-2 as described in Section 2.3). The time complexity is polynomial in the number of input coefficients n and the magnitude of the coefficients c. Since the input size is $L(I) = n \log c$, the running time of $O(nc)$ is not polynomial in the input size. Problems, which can be solved in pseudopolynomial time are also denoted as *weakly $\mathcal{NP}$-hard*.

The most difficult problems are the *strongly $\mathcal{NP}$-hard* problems. This is the class of decision problems which are still $\mathcal{NP}$-hard even when all numbers in the input are bounded by some polynomial in the input size. The *quadratic knapsack problem* in decision form, QKP-DECISION, is strongly $\mathcal{NP}$-hard, since even if all profits and weights are bounded by the constant 1, and c is bounded by n, we may formulate CLIQUE as a QKP-DECISION problem (see Section 12.1). Since CLIQUE is known to be $\mathcal{NP}$-hard, QKP-DECISION must be strongly $\mathcal{NP}$-hard.

For an optimization problem Q we will say that it is $\mathcal{NP}$-hard if the corresponding decision problem is $\mathcal{NP}$-complete. The class of *$\mathcal{NP}$-hard* problems is defined as the set of optimization problems with the given property.

A.2 $\mathcal{NP}$-Completeness of the Subset Sum Problem

To give a more self-contained presentation, we will show that subset sum problem in decision form is harder than any decision problem which efficiently can be formulated as a binary integer linear program. Formally, we define SSP-DECISION as

$$\text{SSP-DECISION}(w,c) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} w_j x_j = c, \\ x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\}. \tag{A.7}$$

A binary integer linear program in decision form is defined as:

$$\text{BIP-DECISION}(W,p,c,t) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} p_j x_j \geq t, \\ \sum_{j=1}^{n} w_{ij} x_j \leq c_i, i = 1,\ldots,m, \\ x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\}, \tag{A.8}$$

where W is an $m \times n$ matrix, c an m-vector, and p an n-vector. We assume that all the coefficients p_j, w_{ij}, c_i are integers.

It is well-known that numerous optimization problems in their decision form can be formulated as BIP-DECISION problems using polynomial time and space, hence allowing a polynomial reduction to BIP-DECISION. Important examples include the *set covering problem*, the *traveling salesman problem*, the *Hamilton cycle problem* and the *formula satisfiability problem* (see Cormen et al. [92] for a definition of these problems).

$$\begin{array}{l} \text{SET-COVERING-DECISION} \leq_p \text{BIP-DECISION}, \\ \text{TSP-DECISION} \leq_p \text{BIP-DECISION}, \\ \text{HAM-CYCLE} \leq_p \text{BIP-DECISION}, \\ \text{FORMULA-SATISFIABILITY} \leq_p \text{BIP-DECISION}. \end{array} \tag{A.9}$$

This gives us a kind of evidence of the hardness of BIP-DECISION. As a matter of fact, BIP-DECISION is $\mathcal{NP}$-complete as shown in e.g. Garey and Johnson [164], and hence all problems in $\mathcal{NP}$ can be reduced to BIP-DECISION.

By noting that the first inequality in (A.8) may be written $\sum_{j=1}^{n} -p_j x_j \leq -t$, we may just consider the objective function as another constraint in the problem. Moreover, adding a number of binary slack variables to each constraint as $x'_{1i} + 2x'_{2i} + 4x'_{3i} + \ldots$ we may reformulate the problem as

$$\text{BIP-DECISION}(W,c) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} w_{ij} x_j = c_i, i = 1,\ldots,m, \\ x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\} \tag{A.10}$$

where $c = (c_1,\ldots,c_m)$ and W is a new matrix but only polynomially larger than the matrix in (A.8). For reasons of simplicity we will consider the latter version of BIP-DECISION.

In order to prove $\mathcal{NP}$-completeness of SSP-DECISION we will first show how to merge two constraints into one, and then use this result to merge all constraints in BIP-DECISION into one constraint in SSP-DECISION in Section A.2.2.

A.2.1 Merging of Constraints

Mathews [338] already in 1897 showed how BIP-DECISION may be transformed to KP-DECISION, hence presenting an early reduction algorithm for $\mathcal{NP}$-complete problems without knowing the present notion of complexity. Consider the following BIP-DECISION problem with two equality constraints

$$\left\{\begin{array}{l}\text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} w_{1j}x_j = c_1, \\ \sum_{j=1}^{n} w_{2j}x_j = c_2, \\ x_j \in \{0,1\},\ j = 1,\ldots,n.\end{array}\right\} \tag{A.11}$$

Let the difference between the right and left side of the constraints be given by

$$\begin{aligned} g(x) &= c_1 - \sum_{j=1}^{n} w_{1j}x_j, \\ h(x) &= c_2 - \sum_{j=1}^{n} w_{2j}x_j. \end{aligned} \tag{A.12}$$

By using the bounds on x_j we derive the following bound on $g(x)$

$$c_1 - \sum_{j=1}^{n} a_j \leq g(x) \leq c_1 - \sum_{j=1}^{n} b_j, \tag{A.13}$$

where $a_j = \max\{w_{1j},0\}$ and $b_j = \min\{w_{1j},0\}$. If we choose a positive integer λ satisfying

$$\lambda > \max\left\{c_1 - \sum_{j=1}^{n} b_j,\ -c_1 + \sum_{j=1}^{n} a_j\right\}, \tag{A.14}$$

then we have $|g(x)| < \lambda$. Now, multiplying the second constraint in (A.11) with λ and adding it to the first constraint, the following problem appears

$$\left\{\begin{array}{l}\text{there exists an } x \text{ with} \\ \sum_{j=1}^{n}(w_{1j} + \lambda w_{2j})x_j = c_1 + \lambda c_2 = \tilde{c}, \\ x_j \in \{0,1\},\ j = 1,\ldots,n\end{array}\right\}. \tag{A.15}$$

Proposition A.2.1 *Equations (A.11) and (A.15) have identical sets of nonnegative integer solutions when λ is chosen according to (A.14).*

Proof. It is evident that (A.11) implies (A.15) for any multiplier λ. Thus to prove that the opposite holds, assume that x is a solution vector to (A.15) with

$$h(x) = K, \tag{A.16}$$

where K must be a (positive or negative) integer, as the coefficients in $h(x)$ are integers. Note, that the constraint in (A.15) may be written

$$g(x) + \lambda h(x) = 0, \tag{A.17}$$

which by insertion of (A.16) gives $g(x) + \lambda K = 0$. But λ was chosen such that $|g(x)| < \lambda$ and hence $|K| < 1$. Since K is an integer we must have $K = 0$. This implies that $h(x) = 0$ and due to (A.17) we also have $g(x) = 0$, so both constraints in (A.11) are satisfied. □

Proposition A.2.2 *The transformation from (A.11) to (A.15) can be performed in polynomial time and space.*

Proof. The only computation made in the transformation is the determination of λ in (A.14) which can be made in linear time. The output of the new problem (A.15) is also made in linear time.

Assuming that all coefficients in (A.11) are bounded by a constant M then the input size of an instance $I \in BIP-DECISION$ is $L(I) = \Theta(n \log M)$ as it consists of $2n$ weights, and the parameters c_1, c_2, taking up space $2n \log M + 2 \log M$.

Since $\lambda \leq nM$, all the weights (and the capacity) in the output are bounded by nM^2. Hence the output size becomes $\Theta(n \log(nM))$ since we have $n+1$ weights of size $\log(nM^2)$, and one capacity of similar size.

Since $\Theta(n \log(nM))$ is polynomial in $n \log M$ we have shown the stated. □

Notice that a similar technique was also used in the proof of Theorem 8.1.2.

A.2.2 $\mathcal{NP}$-Completeness

We are now ready to show:

Proposition A.2.3 SSP-DECISION *is $\mathcal{NP}$-complete.*

Proof. Using similar arguments as in Lemma A.1.1 we note that $SSP-DECISION \in \mathcal{NP}$. To show that all problems $\forall R \in \mathcal{NP} : R \leq_p KP-DECISION$ we choose to reduce from the $\mathcal{NP}$-complete problem BIP-DECISION. Hence assume that an instance of BIP-DECISION is given by (A.10). If we repeatedly merge the first constraint with each of the constraints $i = 2, \ldots, m$ we obtain an instance of SSP-DECISION. Due to Proposition A.2.1 we have that a "yes" instance will be transformed to a "yes" instance and vice versa. The time and space complexity of the transformation algorithm is polynomial since it is a concatenation of $m-1$ transformation algorithms each having polynomial time and space complexity. This shows that

$$\text{BIP-DECISION} \leq_p \text{SSP-DECISION}$$

which gives the stated. □

It is interesting to analyze the actual time and space complexity of the transformation algorithm. Since all weights are bounded by the capacity, it is sufficient to analyze the magnitude of the transformed capacity. Assuming that all coefficients in the BIP-DECISION problem are bounded by M, the transformed capacity $\tilde{c}$ in (A.15) will be bounded by nM^2 after merging two constraints. Having merged m constraints, the transformed capacity will be bounded by $\tilde{c} \leq n^{m-1} M^m$. Thus, the magnitude of the capacity will grow exponentially with the number of constraints m.

We have previously noticed that SSP-DECISION can be decided in pseudopolynomial time $O(nc)$ through dynamic programming. Using the above reduction of BIP-DECISION to SSP-DECISION and then solving the problem through dynamic programming will however not result in a pseudopolynomial algorithm. Due to the exponential blow-up of the capacity, the resulting algorithm will run in $O(n\tilde{c}) = O(nn^{m-1}M^m) = O(n^mM^m)$ which is not polynomial in the number of input coefficients $n \times m$ and the magnitude M of the coefficients.

A.3 $\mathcal{NP}$-Completeness of the Knapsack Problem

The KP-DECISION problem was formally defined in (A.2). We will now prove that

Proposition A.3.1 KP-DECISION *is $\mathcal{NP}$-complete.*

Proof. We noticed in Lemma A.1.1 that KP-DECISION is in $\mathcal{NP}$. To show the $\mathcal{NP}$-hardness of KP-DECISION we reduce from SSP-DECISION. Hence assume that an instance of the latter problem is given. Setting $p_j = w_j$ for $j = 1,\ldots,n$, and choosing $t = c$ we notice that SSP-DECISION has a feasible solution if and only the corresponding KP-DECISION has a feasible solution. The reduction obviously runs in polynomial time. □

A.4 $\mathcal{NP}$-Completeness of Other Knapsack Problems

It is easy to show the $\mathcal{NP}$-completeness of the more general knapsack problems, like the *bounded knapsack problem*, *multiple knapsack problem* and the *multidimensional knapsack problem*, as all these problems contain KP-DECISION as a special case.

Some of the less trivial examples include the *unbounded subset sum problem*, the *unbounded knapsack problem*, the *multiple-choice knapsack problem*, and the *quadratic knapsack problem*.

We define formally USSP-DECISION as the problem

$$\text{USSP-DECISION}(w,c) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} w_j x_j = c, \\ x_j \in \mathbb{N}_0, j = 1,\ldots,n \end{array} \right\} \tag{A.18}$$

Proposition A.4.1 USSP-DECISION *is $\mathcal{NP}$-complete.*

Proof. Obviously $USSP-DECISION \in \mathcal{NP}$, as can be shown by use of the same arguments as in Lemma A.1.1. To show the $\mathcal{NP}$-hardness of the problem, we reduce from SSP-DECISION. Adding slack variables, SSP-DECISION may be formulated equivalently as

$$\text{SSP-DECISION}(w,c) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} w_j x_j = c, \\ x_j + y_j = 1, j = 1,\ldots,n \\ x_j, y_j \in \mathbb{N}_0, j = 1,\ldots,n \end{array} \right\} \tag{A.19}$$

We will use Proposition A.2.1 for merging the constraints in SSP-DECISION. Consider the two equations

$$\begin{aligned} x_1 + y_1 &= 1, \\ \sum_{j=1}^{n} w_j x_j &= c. \end{aligned} \tag{A.20}$$

As stated in (A.14), we must choose a λ satisfying

$$\lambda > \max\left\{ 1 - \sum_{j=1}^{2} 0,\ -1 + \sum_{j=1}^{2} 1 \right\} = 1, \tag{A.21}$$

Using $\lambda = 2$ we reach the equivalent equation

$$x_1 + y_1 + 2\sum_{j=1}^{n} w_j x_j = 2c + 1. \tag{A.22}$$

In the next iteration we merge the constraints $x_2 + y_2 = 1$ and (A.22). As before, we may use $\lambda = 2$ in Proposition A.2.1 getting the equivalent equation

$$x_2 + y_2 + 2x_1 + 2y_1 + 2^2 \sum_{j=1}^{n} w_j x_j = 2(2c+1) + 1. \tag{A.23}$$

Repeating the process of merging with $x_j + y_j = 1$ for $j = 3,\ldots,n$ we get

$$\sum_{j=1}^{n} 2^{n-j} y_j + \sum_{j=1}^{n} 2^{n-j} x_j + 2^n \sum_{j=1}^{n} w_j x_j = 2^n c + 2^n - 1. \tag{A.24}$$

Setting $\tilde{w}_j = 2^{n-j} + 2^n w_j$, $\overline{w}_j = 2^{n-j}$ and $\bar{c} = 2^n c + 2^n - 1$, we reach the equivalent formulation

$$\left\{ \begin{array}{l} \text{there exists } x, y \text{ with} \\ \sum_{j=1}^{n} \tilde{w}_j x_j + \sum_{j=1}^{n} \overline{w}_j y_j = \bar{c}, \\ x_j, y_j \in \mathbb{N}_0, j = 1,\ldots,n \end{array} \right\} \tag{A.25}$$

Hence, we have reached an instance equivalent to USSP-DECISION. Since the reduction is polynomial in time and space, we have proved the desired. □

The *unbounded knapsack problem* in decision form UKP-DECISION is defined as

$$\text{UKP-DECISION}(p,w,c,t) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} p_j x_j \geq t, \\ \sum_{j=1}^{n} w_j x_j \leq c, \\ x_j \in \mathbb{N}_0, j = 1,\ldots,n \end{array} \right\} \tag{A.26}$$

Proposition A.4.2 UKP-DECISION *is* $\mathcal{NP}$*-complete.*

Proof. UKP-DECISION is obviously in $\mathcal{NP}$. We prove the $\mathcal{NP}$-hardness by reduction from USSP-DECISION. For a given instance of USSP-DECISION, we set $p_j = w_j$ for $j = 1,\ldots,n$ and the threshold value is set to $t = c$. □

The *multiple-choice knapsack problem* in decision form MCKP-DECISION is the problem

$$\text{MCKP-DECISION}(p,w,c,t) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{i=1}^{m} \sum_{j=1}^{n_i} p_{ij} x_{ij} \geq t, \\ \sum_{i=1}^{m} \sum_{j=1}^{n_i} w_{ij} x_{ij} \leq c, \\ \sum_{j=1}^{n_i} x_{ij} = 1,\ i = 1,\ldots,m \\ x_{ij} \in \{0,1\},\ i = 1,\ldots,m, \\ \qquad\qquad j = 1,\ldots,n_i \end{array} \right\} \tag{A.27}$$

Proposition A.4.3 MCKP-DECISION *is* $\mathcal{NP}$*-complete.*

Proof. It is obvious that MCKP-DECISION is in $\mathcal{NP}$. We will prove the $\mathcal{NP}$-hardness by reduction from KP-DECISION. For an instance of KP-DECISION construct an equivalent instance of MCKP-DECISION by introducing n classes, each having two items. The profit and weight of the two items is $(\tilde{p}_{j1}, \tilde{w}_{j1}) = (0,0)$ respectively $(\tilde{p}_{j2}, \tilde{w}_{j2}) = (p_j, w_j)$ for $j = 1,\ldots,n$. □

Finally we consider the *quadratic knapsack problem* in decision form, given by:

$$\text{QKP-DECISION}(p,w,c,t) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} x_i x_j \geq t, \\ \sum_{j=1}^{n} w_j x_j \leq c, \\ x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\}. \tag{A.28}$$

Proposition A.4.4 QKP-DECISION *is* $\mathcal{NP}$*-complete.*

Proof. Given an instance of KP-DECISION. Construct an equivalent instance of QKP-DECISION by defining the profit matrix $\tilde{P} = (\tilde{p_{ij}})$ of the latter problem as $\tilde{p}_{jj} = p_j$ for $j = 1,\ldots,n$ and $\tilde{p}_{ij} = 0$ for $i \neq j$. The values c and t are unchanged. □

References

1. E. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

2. F.B. Abdelaziz, S. Krichen, and J. Chaouachi. A hybrid heuristic for multiobjective knapsack problems. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 205–212. Kluwer, 1999.

3. W.P. Adams and H.D. Sherali. A tight linearization and an algorithm for zero-one quadratic programming problems. *Management Science*, 32:1274–1290, 1986.

4. L.M. Adleman. On breaking generalized knapsack public key cryptosystems. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 402–412, 1983.

5. M. Agrawal, E. Allender, and S. Datta. On TC^0, AC^0, and arithmetic circuits. Technical Report 97-48, DIMACS, 1997.

6. J.H. Ahrens and G. Finke. Merging and sorting applied to the zero-one knapsack problem. *Operations Research*, 23:1099–1109, 1975.

7. L. Aittoniemi and K. Oehlandt. A note on the Martello-Toth algorithm for one-dimensional knapsack problems. *European Journal of Operational Research*, 20:117, 1985.

8. M.M. Akbar, E.G. Manning, G.C. Shoja, and S. Khan. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *Computational Science (ICCS 2001)*, volume 2074 of *Lecture Notes in Computer Science*, pages 659–668. Springer, 2001.

9. R. Alvarez-Valdes, A. Parajon, and J.M. Tamarit. A computational study of heuristic algorithms for two-dimensional cutting stock problems. In *Proceedings of 4th Meta-heuristics International Conference (MIC 2001)*, pages 7–11, 2001.

10. M.J. Alves and J. Climacao. An interactive method for 0-1 multiobjective problems using simulated annealing and tabu search. *Journal of Heuristics*, 6:385–403, 2000.

11. R. Andonov, V. Poirriez, and S. Rajopadhye. Unbounded knapsack problem: dynamic programming revisited. *European Journal of Operational Research*, 123:394–407, 2000.

12. R. Andonov and S. Rajopadhye. A sparse knapsack algo-tech-cuit and its synthesis. In *International Conference on Application Specific Array Processors ASPA '94*. IEEE, 1994.

13. A. Archer, C.H. Papadimitriou, K. Talwar, and E. Tardos. An approximate truthful mechanism for combinatorial auctions with single parameter agents. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, 2003.

14. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer, 1999.

15. I. Averbakh. Probabilistic properties of the dual structure of the multidimensional knapsack problem and fast statistically efficient algorithms. *Mathematical Programming*, 65:311–330, 1994.

16. D.A. Babayev, F. Glover, and J. Ryan. A new knapsack solution approach by integer equivalent aggregation and consistency determination. *INFORMS Journal on Computing*, 9:43–50, 1997.

17. D.A. Babayev and S. Mardanov. Reducing the number of variables in integer and linear programming problems. *Computational Optimization and Applications*, 3:99–109, 1994.

18. E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

19. E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

20. E. Balas and C.H. Martin. Pivot and complement - a heuristic for 0-1 programming. *Management Science*, 26:86–96, 1980.

21. E. Balas and E. Zemel. Facets of the knapsack polytope from minimal covers. *SIAM Journal of Applied Mathematics*, 34:119–148, 1978.

22. E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28:1130–1154, 1980.

23. E. Balas and E. Zemel. Lifting and complementing yields all the facets of positive zero-one programming polytopes. In R.W. Cottle, M.L. Kelmanson, and B. Korte, editors, *Mathematical Programming*, pages 13–24. Elsevier, 1984.

24. S. Balev, N. Yanev, A. Fréville, and R. Andonov. A dynamic programming based reduction procedure for the multidimensional 0-1 knapsack problem. Technical report, University of Valenciennes, 2001.

25. B. Bank and R. Mandel. *Parametric Integer Optimization*. Akademie-Verlag, 1988.

26. P. Barcia and K. Jörnsten. Improved Lagrangean decomposition: an application to the generalized assignment problem. *European Journal of Operational Research*, 46:84–92, 1990.

27. R.S. Barr, B.L. Golden, J.P. Kelly, M.G.C. Resende, and W.R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1:9–32, 1995.

28. R. Battiti and G. Tecchiolli. Local search with memory: benchmarking RTS. *OR Spectrum*, 17:67–86, 1995.

29. M. Bauvin and M.X. Goemans. Personal communication, 1999.

30. R. Beier and B. Vöcking. Random knapsack in expected polynomial time. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, 2003.

31. L. Beinsen and U. Pferschy. Economic scenarios involving problems of the knapsack family. manuscript in preparation, Faculty of Economics, University of Graz.

32. R.E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

33. M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 80–86, 1983.

34. D. Bertsimas and R. Demir. An approximate dynamic programming approach to multidimensional knapsack problems. *Management Science*, 48:550–565, 2002.

35. K. Bhaskar. A multiple objective approach to capital budgeting. *Accounting and Business Research*, 9:25–46, 1979.

36. A. Billionnet. Approximation algorithms for fractional knapsack problems. *Operations Research Letters*, 30:336–342, 2002.

37. A. Billionnet and F. Calmels. Linear programming for the 0-1 quadratic knapsack problem. *European Journal of Operational Research*, 92:310–325, 1996.

38. A. Billionnet, A. Faye, and E. Soutif. A new upper bound and an exact algorithm for the 0-1 quadratic knapsack problem. *European Journal of Operational Research*, 112:664–672, 1999.

39. L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer, 1997.

40. M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

41. K.H. Borgwardt and J. Brzank. Average saving effects in enumerative methods for solving knapsack problems. *Journal of Complexity*, 10:129–141, 1994.

42. K.H. Borgwardt and B. Tremel. The average quality of greedy-algorithms for the subset-sum-maximization problem. *ZOR - Methods and Models of Operations Research*, 35:113–149, 1991.

43. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

44. E.A. Boyd. Polyhedral results for the precedence-constrained knapsack problem. *Discrete Applied Mathematics*, 41:185–201, 1993.

45. S.P. Bradley, A.C. Hax, and T.L. Magnanti. *Applied Mathematical Programming*. Addison Wesley, 1977.

46. A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes: a Survey*. SIAM Monographs on Discrete Mathematics and Applications, 1999.

47. K.M. Bretthauer and B. Shetty. The nonlinear resource allocation problem. *Operations Research*, 43:670–683, 1995.

48. K.M. Bretthauer and B. Shetty. The nonlinear knapsack problem – algorithms and applications. *European Journal of Operational Research*, 138:459–472, 2002.

49. K.M. Bretthauer and B. Shetty. A pegging algorithm for the nonlinear resource allocation problem. *Computers and Operations Research*, 29:505–527, 2002.

50. K.M. Bretthauer, B. Shetty, and S. Syam. A branch-and-bound algorithm for integer quadratic knapsack problems. *ORSA Journal on Computing*, 7:109–116, 1995.

51. E.F. Brickell. Solving low density knapsacks. In *Advances in Cryptology – CRYPTO '83*, pages 24–37. Plenum Press, 1984.

52. E.F. Brickell. Breaking iterated knapsacks. In *Advances in Cryptology – CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 1985.

53. E.F. Brickell, J.C. Lagarias, and A.M. Odlyzko. Evaluation of the Adleman attack on multiple iterated knapsack cryptosystems. In *Advances in Cryptology – CRYPTO '83*, pages 39–42. Plenum Press, 1984.

54. E.F. Brickell and A.M. Odlyzko. Cryptanalysis: a survey of recent results. In G.J. Simmons, editor, *Contemporary Cryptology*, pages 501–540. IEEE Press, 1991.

55. V.E. Brimkov and S.S. Dantchev. An alternative to Ben-Or's lower bound for the knapsack problem complexity. *Applied Mathematics Letters*, 15:187–191, 2002.

56. J.R. Brown. The knapsack sharing problem. *Operations Research*, 27:341–355, 1979.

57. R.L. Bulfin, R.G. Parker, and C.M. Shetty. Computational results with a branch-and-bound algorithm for the general knapsack problem. *Naval Research Logistics Quarterly*, 26:41–46, 1979.

58. R.E. Burkard and U. Pferschy. The inverse-parametric knapsack problem. *European Journal of Operational Research*, 83:376–393, 1995.

59. A.V. Cabot. An enumeration algorithm for knapsack problems. *Operations Research*, 18:306–311, 1970.

60. G.J. Caesar. *Commentarii de Bello Gallico*. Rome, (51 B.C.).

61. J.M. Calvin and J.Y-T. Leung. Average-case analysis of a greedy algorithm for the 0/1 knapsack problem. *Operations Research Letters*, 31:202–210, 2003.

62. A. Caprara, H. Kellerer, and U. Pferschy. The multiple subset sum problem. *SIAM Journal of Optimization*, 11:308–319, 2000.

63. A. Caprara, H. Kellerer, and U. Pferschy. A PTAS for the multiple subset sum problem with different knapsack capacities. *Information Processing Letters*, 73:111–118, 2000.

64. A. Caprara, H. Kellerer, and U. Pferschy. A 3/4 -approximation algorithm for multiple subset sum. *Journal of Heuristics*, 9:99–111, 2003.

65. A. Caprara, H. Kellerer, U. Pferschy, and D. Pisinger. Approximation algorithms for knapsack problems with cardinality constraints. *European Journal of Operational Research*, 123:333–345, 2000.

66. A. Caprara, A. Lodi, and M. Monaci. An approximation scheme for the two-stage, two-dimensional bin packing problem. In *Integer Programming and Combinatorial Optimization, 9th IPCO Conference*, volume 2337 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2002.

67. A. Caprara and U. Pferschy. Packing bins with minimal slack. Technical Report 02/2002, Faculty of Economics, University of Graz, 2002.

68. A. Caprara and U. Pferschy. Worst-case analysis of the subset sum algorithm for bin packing. *Operations Research Letters*, 2003. to appear.

69. A. Caprara, D. Pisinger, and P. Toth. Exact solution of the quadratic knapsack problem. *INFORMS Journal on Computing*, 11:125–137, 1999.

70. M.E. Captivo, J. Climacao, J. Figueira, E. Martins, and J.L. Santos. Solving bicriteria 0-1 knapsack problems using a labeling algorithm. *Computers and Operations Research*, 30:1865–1886, 2003.

71. P. Carraresi and F. Malucelli. A reformulation scheme and new lower bounds for the QAP. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, pages 147–160. AMS Press, 1994.

72. R.L. Carraway, R.L. Schmidt, and L.R. Weatherford. An algorithm for maximizing target achievement in the stochastic knapsack problem with normal returns. *Naval Research Logistics*, 40:161–173, 1993.

73. P.J. Carstensen. Complexity of some parametric integer and network programming problems. *Mathematical Programming*, 26:64–75, 1983.

74. J.O. Cerdeira and P. Barcia. When is a 0-1 knapsack a matroid? *Portugaliae Mathematica*, 52:475–480, 1995.

75. P. Chaillou, P. Hansen, and Y. Mahieu. Best network flow bound for the quadratic knapsack problem. In B. Simeone, editor, *Combinatorial Optimization*, volume 1403 of *Lecture notes in mathematics*, pages 225–235. Springer, 1986.

76. E.D. Chajakis and M. Guignard. Exact algorithms for the setup knapsack problem. *INFOR*, 32:124–142, 1994.

77. A.K. Chandra, D.S. Hirschberg, and C.K. Wong. Approximate algorithms for some generalized knapsack problems. *Theoretical Computer Science*, 3:293–304, 1976.

78. S.K. Chang and A. Gill. Algorithmic solution of the change-making problem. *Journal of the ACM*, 17:113–122, 1970.

79. C. Chekuri and S. Khanna. A PTAS for the multiple knapsack problem. In *Proceedings of the 11th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 213–222, 2000.

80. G. Cho and D.X. Shaw. A depth-first dynamic programming algorithm for the tree knapsack problem. *INFORMS Journal on Computing*, 9:431–438, 1997.

81. B. Chor and R.L. Rivest. A knapsack type public key cryptosystem based on arithmetic in finite fields. *IEEE Transactions on Information Theory*, IT-34:901–999, 1988.

82. N. Christofides, A. Mingozzi, and P. Toth. Loading problems. In N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, pages 339–369. Wiley, Chichester, 1979.

83. P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

84. K.L. Chung. *A Course in Probability Theory*. Academic Press, second edition, 2000.

85. V. Chvátal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

86. V. Chvátal. Hard knapsack problems. *Operations Research*, 28:1402–1411, 1980.

87. C.A. Coello Coello. http://www.lania.mx/~ccoello/EMOO/EMOOjournals.html.

88. E.G. Coffman and G.S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms*. J. Wiley, 1991.

89. A.M. Cohn and C. Barnhart. The stochastic knapsack problem with random weights: a heuristic approach to robust transportation planning. In *Proceedings of the Triennial Symposium on Transportation Analysis (TRISTAN III)*, 1998.

90. S. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

91. S. Cook. The P versus NP problem, 2000. Clay Mathematics Institute, Millenium Prize Problems.

92. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

93. Y. Crama and J.B. Mazzola. On the strength of relaxations of multidimensional knapsack problems. *INFOR*, 32:219–225, 1994.

94. P. Crescenzi and V. Kann. A compendium of NP optimization problems. http://www.nada.kth.se/~viggo/problemlist/compendium.html.

95. J. Csirik, H. Kellerer, and G.J. Woeginger. The exact LPT method for maximizing the minimum completion time. *Operations Research Letters*, 11:281–287, 1992.

96. P. Czyzak and A. Jaszkiewicz. Pareto simulated annealing - a metaheuristic technique for multiple objective combinatorial optimization. *Journal of Multi-Criteria Decision Analysis*, 7:34–47, 1998.

97. G.B. Dantzig. Discrete variable extremum problems. *Operations Research*, 5:266–277, 1957.

98. G. d'Atri and A. di Rende. Probabilistic analysis of knapsack-type problems. *Methods of Operations Research*, 40:279–282, 1980.

99. G. d'Atri and C. Puech. Probabilistic analysis of the subset-sum problem. *Discrete Applied Mathematics*, 4:329–334, 1982.

100. M. Dawande, J. Kalagnanam, P. Keskinocak, R. Ravi, and F.S. Salman. Approximation algorithms for the multiple knapsack problem with assignment restrictions. *Journal of Combinatorial Optimization*, 4:171–186, 2000.

101. I.R. de Farias and G.L. Nemhauser. A polyhedral study of the cardinality constrained knapsack polytope. In *Integer Programming and Combinatorial Optimization, 9th IPCO conference*, volume 2337 of *Lecture Notes in Computer Science*, pages 291–303. Springer, 2002. to be published in *Mathematical Programming*.

102. R. De Leone, R. Jain, and K. Straus. Solution of multiple-choice knapsack problem encountered in high-level synthesis of VLSI circuits. *Journal of Computer Mathematics*, 47:163–176, 1993.

103. S. de Vries and R.V. Vohra. Combinatorial auctions: a survey. *INFORMS Journal on Computing*, 15:284–309, 2003.

104. R.S. Dembo and P.L. Hammer. A reduction algorithm for knapsack problems. *Methods of Operations Research*, 36:49–60, 1980.

105. B.L. Dietrich and L.F. Escudero. Coefficient reduction for knapsack constraints in 0-1 programs with VUB's. *Operations Research Letters*, 9:9–14, 1990.

106. G. Dijkhuizen and U. Faigle. A cutting-plane approach to the edge-weighted maximal clique problem. *European Journal of Operational Research*, 69:121–130, 1993.

107. W. Dinkelbach. On nonlinear fractional programming. *Management Science*, 13:492–498, 1967.

108. G. Diubin and A. Korbut. On the average behaviour of greedy algorithms for the knapsack problem. Technical Report 99-14, Humboldt University Berlin, Department of Mathematics, 1999.

109. G. Diubin and A. Korbut. The average behaviour of greedy algorithms for the knapsack problem: general distributions. Technical Report 00-19, Humboldt University Berlin, Department of Mathematics, 2000.

110. D. Dobkin and R.J. Lipton. A lower bound of $(1/2)n^2$ on linear search programs for the knapsack problem. *Journal of Computer and System Sciences*, 16:413–417, 1978.

111. G. Dobson. Worst-case analysis of greedy heuristics for integer programming with nonnegative data. *Mathematics of Operations Research*, 7:515–531, 1982.

112. D. Dor and U. Zwick. Selecting the median. *SIAM Journal on Computing*, 28:1722–1758, 1999.

113. A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, 40:1–8, 1988.

114. K. Dudziński. A note on dominance relation in unbounded knapsack problems. *Operation Research Letters*, 10:417–419, 1991.

115. K. Dudziński and S. Walukiewicz. A fast algorithm for the linear multiple-choice knapsack problem. *Operations Research Letters*, 3:205–209, 1984.

116. K. Dudziński and S. Walukiewicz. On the multiperiod binary knapsack problem. *Methods of Operations Research*, 49:223–232, 1985.

117. K. Dudziński and S. Walukiewicz. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*, 28:3–21, 1987.

118. J. Dupacova, J. Hurt, and J. Stepan. *Stochastic Modeling in Economics and Finance*. Kluwer, 2002.

119. H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.

120. H. Dyckhoff and U. Finke. *Cutting and Packing in Production and Distribution*. Physica, 1992.

121. M.E. Dyer. Calculating surrogate constraints. *Mathematical Programming*, 19:255–278, 1980.

122. M.E. Dyer. An $O(n)$ algorithm for the multiple-choice knapsack linear program. *Mathematical Programming*, 29:57–63, 1984.

123. M.E. Dyer and A.M. Frieze. Probabilistic analysis of the multidimensional knapsack problem. *Mathematics of Operations Research*, 14:162–176, 1989.

124. M.E. Dyer, N. Kayal, and J. Walker. A branch and bound algorithm for solving the multiple choice knapsack problem. *Journal of Computational and Applied Mathematics*, 11:231–249, 1984.

125. M.E. Dyer, W.O. Riha, and J. Walker. A hybrid dynamic programming/branch-and-bound algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 58:43–54, 1995.

126. M.E. Dyer and J. Walker. Dominance in multi-dimensional multiple-choice knapsack problems. *Asia-Pacific Journal of Operational Research*, 15:159–168, 1998.

127. M. Eben-Chaime. Parametric solution for linear bicriteria knapsack models. *Management Science*, 42:1565–1575, 1996.

128. M. Ehrgott and X. Gandibleux, editors. *Multiple Criteria Optimization: State of the Art Annotated Bibliographical Surveys*. Kluwer, 2002.

129. S. Eilon and N. Christofides. The loading problem. *Management Science*, 17:259–268, 1971.

130. E.J. Elton and M.J. Gruber. *Modern Portfolio Theory and Investment Analysis*. J. Wiley, 5th edition, 1995.

131. J.B. Epstein, P.G. Epstein, and H. Koch. *Casablanca*. Warner/First National, 1942.

132. T. Erlebach, H. Kellerer, and U. Pferschy. Approximating multi-objective knapsack problems. *Management Science*, 48:1603–1612, 2002.

133. L.F. Escudero, A. Garín, and G. Pérez. An $O(n \log n)$ procedure for identifying facets of the knapsack polytope. *Operations Research Letters*, 31:211–218, 2003.

134. L.F. Escudero, S. Martello, and P. Toth. A framework for tightening 0-1 programs based on an extension of pure 0-1 KP and SS problems. In *Integer Programming and Combinatorial Optimization, 4th IPCO conference*, volume 920 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 1995.

135. H. Everett. Generalized Lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11:399–417, 1963.

136. B.H. Faaland. The multiperiod knapsack problem. *Operations Research*, 29:612–616, 1981.

137. D. Fayard and G. Plateau. Reduction algorithm for single and multiple constraints 0-1 linear programming problems. In *Conference on Methods of Mathematical Programming*, Zakopane (Poland), 1977.

138. D. Fayard and G. Plateau. An algorithm for the solution of the 0-1 knapsack problem. *Computing*, 28:269–287, 1982.

139. D. Fayard and G. Plateau. An exact algorithm for the 0–1 collapsing knapsack problem. *Discrete Applied Mathematics*, 49:175–187, 1994.

140. D. Fayard and V. Zissimopoulos. An approximation algorithm for solving unconstrained two-dimensional knapsack problems. *European Journal of Operational Research*, 84:618–632, 1995.

141. W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. J. Wiley, 1968.

142. W. Fernandez de la Vega and G.S. Lueker. Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica*, 1:349–355, 1981.

143. C.E. Ferreira, A. Martin, and R. Weismantel. Solving multiple knapsack problems by cutting planes. *SIAM Journal on Optimization*, 6:858–877, 1996.

144. M. Feuerman and H. Weiss. A mathematical programming model for test construction and scoring. *Management Science*, 19:961–966, 1973.

145. A. Fiat and G.J. Woeginger, editors. *Online Algorithms: the State of the Art*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998.

146. M. Fischetti. A new linear storage polynomial-time approximation scheme for the subset-sum problem. *Discrete Applied Mathematics*, 26:61–77, 1990.

147. M.L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27:1–18, 1981.

148. J.F. Fontanari. A statistical analysis of the knapsack problem. *Journal of Physics A*, 28:4751–4759, 1995.

149. H. Fournier and P. Koiran. Are lower bounds easier over the reals? In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 507–513, 1998.

150. G.E. Fox and G.D. Scudder. A heuristic with tie breaking for certain 0–1 integer programming models. *Naval Research Logistics Quarterly*, 32:613–623, 1985.

151. M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

152. A. Fréville and G. Plateau. Heuristic and reduction methods for multiple constraints 0–1 linear programming problems. *European Journal of Operational Research*, 24:206–215, 1986.

153. A. Fréville and G. Plateau. An exact search for the solution of the surrogate dual for the 0-1 bidimensional knapsack problem. *European Journal of Operational Research*, 68:413–421, 1993.

154. A. Fréville and G. Plateau. An efficient preprocessing procedure for the multidimensional knapsack problem. *Discrete Applied Mathematics*, 49:189–212, 1994.

155. A. Fréville and G. Plateau. The 0-1 bidimensional knapsack problem: towards an efficient high-level primitive tool. *Journal of Heuristics*, 2:147–167, 1996.

156. A.M. Frieze and M.R.B. Clarke. Approximation algorithms for the m-dimensional 0–1 knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 15:100–109, 1984.

157. A.M. Frieze and B. Reed. Probabilistic analysis of algorithms. In M. Habib et al., editor, *Probabilistic Methods for Algorithmic Discrete Mathematics*. Springer, 1998.

158. K. Fujisawa, M. Kojima, and K. Nakata. *SDPA Semidefinite Programming Algorithm*. Tokyo Institute of Technology, Japan, 1999.

159. G. Gallo, M. Grigoriadis, and R.E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18:30–55, 1989.

160. G. Gallo, P.L. Hammer, and B. Simeone. Quadratic knapsack problems. *Mathematical Programming Study*, 12:132–149, 1980.

161. X. Gandibleux and A. Fréville. Tabu search based procedure for solving the 0-1 multiobjective knapsack problem: the two objective case. *Journal of Heuristics*, 6:361–383, 2000.

162. X. Gandibleux, N. Mezdaoui, and A. Fréville. A tabu search procedure to solve multiobjective combinatorial optimization problems. In R. Caballero, F. Ruiz, and R. Steuer, editors, *Advances in Multiple Objective and Goal Programming*, volume 455 of *Lecture Notes in Economics and Mathematical Systems*, pages 291–300. Springer, 1997.

163. X. Gandibleux, H. Morita, and N. Katoh. The supported solutions used as a genetic information in a population heuristic. In *First International Conference on Evolutionary Multi-Criterion Optimization*, volume 1993 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2001.

164. M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.

165. B. Gavish and H. Pirkul. Allocation of data bases and processors in a distributed computing system. In J. Akoka, editor, *Management of Distributed Data Processing*, pages 215–231. North-Holland, 1982.

166. B. Gavish and H. Pirkul. Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality. *Mathematical Programming*, 31:78–105, 1985.

167. G.V. Gens and E.V. Levner. Approximation algorithms for certain universal problems in scheduling theory. *Soviet Journal of Computers and System Sciences*, 6:31–36, 1978.

168. G.V. Gens and E.V. Levner. Computational complexity of approximation algorithms for combinatorial problems. In *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 292–300. Springer, 1979.

169. G.V. Gens and E.V. Levner. Fast approximation algorithms for knapsack type problems. In K. Iracki, K. Malinowski, and S. Walukiewicz, editors, *Optimization Techniques, Part 2*, volume 74 of *Lecture Notes in Control and Information Sciences*, pages 185–194. Springer, 1980.

170. G.V. Gens and E.V. Levner. A fast approximation algorithm for the subset-sum problem. *INFOR*, 32:143–148, 1994.

171. G.V. Gens and E.V. Levner. An approximate binary search algorithm for the multiple-choice knapsack problem. *Information Processing Letters*, 67:261–265, 1998.

172. A.M. Geoffrion. Lagrangian relaxation for integer programming. *Mathematical Programming Study*, 2:82–114, 1974.

173. D. Ghosh and N. Chakravarti. Hard knapsack problems that are easy for local search. *Asia-Pacific Journal of Operational Research*, 16:165–172, 1999.

174. P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.

175. P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting stock problem, part II. *Operations Research*, 11:863–888, 1963.

176. P.C. Gilmore and R.E. Gomory. Multistage cutting stock problems of two and more dimensions. *Operations Research*, 13:94–120, 1964.

177. P.C. Gilmore and R.E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14:1045–1074, 1966.

178. F. Glover. A multiphase dual algorithm for the zero-one integer programming problem. *Operations Research*, 13:879–919, 1965.

179. F. Glover. Surrogate constraints. *Operations Research*, 16:741–749, 1968.

180. F. Glover. Integer programming over a finite additive group. *SIAM Journal on Control*, 7:213–231, 1969.

181. F. Glover. Surrogate constraint duality in mathematical programming. *Operations Research*, 23:434–451, 1975.

182. F. Glover and G.A. Kochenberger. Critical event tabu search for multidimensional knapsack problems. In I.H. Osman and J.P. Kelly, editors, *Meta-Heuristics: Theory & Applications*. Kluwer, 1996.

183. F. Glover and G.A. Kochenberger. Solving quadratic knapsack problems by reformulation and tabu search. single constraint case. In P. M. Pardalos, A. Migdalas, and R. E. Burkard, editors, *Combinatorial and Global Optimization*. World Scientific, 2002.

184. F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

185. A. Goel and P. Indyk. Stochastic load balancing and related problems. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, pages 579–586, 1999.

186. A.V. Goldberg and A. Marchetti-Spaccamela. On finding the exact solution of a zero-one knapsack problem. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC)*, pages 359–368, 1984.

187. O. Goldschmidt, D. Nehme, and G. Yu. Note: On the set-union knapsack problem. *Naval Research Logistics*, 41:833–842, 1994.

188. L. Golubchik, S. Khanna, S. Khuller, R. Thurimella, and A. Zhu. Approximation algorithms for data placement on parallel disks. In *Proceedings of the 11th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 223–232, 2000.

189. R. Gonen and D. Lehmann. Optimal solutions for multi-unit combinatorial auctions: branch-and-bound heuristics. In *Proceedings of the 2nd ACM Conference on Electronic Commerce (EC-00)*, pages 13–20, 2000.

190. R. Gonen and D. Lehmann. Linear programming helps solving large multi-unit combinatorial auctions. unpublished manuscript, 2001.

191. J. Gottlieb. On the effectivity of evolutionary algorithms for the multidimensional knapsack problem. In *Artificial Evolution*, volume 1829 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2000.

192. J. Gottlieb. Permutation-based evolutionary algorithms for multidimensional knapsack problems. In *Proceedings of the ACM Symposium on Applied Computing*, pages 415–421, 2000.

193. H.J. Greenberg. An algorithm for the periodic solutions in the knapsack problem. *Journal of Mathematical Analysis and Applications*, 111:327–331, 1985.

194. H.J. Greenberg. On equivalent knapsack problems. *Discrete Applied Mathematics*, 14:263–268, 1986.

195. H.J. Greenberg. An annotated bibliography for post-solution analysis in mixed integer programming and combinatorial optimization. In D.L. Woodruff, editor, *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search*, pages 97–148. Kluwer, 1998.

196. H.J. Greenberg and I. Feldman. A better-step-off algorithm for the knapsack problem. *Discrete Applied Mathematics*, 2:21–25, 1980.

197. H.J. Greenberg and R.L. Hegerich. A branch search algorithm for the knapsack problem. *Management Science*, 16:327–332, 1970.

198. H.J. Greenberg and W.P. Pierskalla. Surrogate mathematical programming. *Operations Research*, 18:924–939, 1970.

199. M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and combinatorics*. Springer, 1988.

200. C. Guéret and C. Prins. A new lower bound for the open-shop problem. *Annals of Operations Research*, 92:165–183, 1999.

201. M.M. Güntzer and D. Jungnickel. Approximate minimization algorithms for the 0/1 knapsack and subset-sum problem. *Operations Research Letters*, 26:55–66, 2000.

202. M.M. Güntzer, D. Jungnickel, and M. Leclerc. Efficient algorithms for the clearing of interbank payments. *European Journal of Operational Research*, 106:212–219, 1998.

203. M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed, editors. *Probabilistic Methods for Algorithmic Discrete Mathematics*. Springer, 1998.

204. E. Hadjiconstantinou and N. Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83:39–56, 1995.

205. T. Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer, 1998.

206. P.L. Hammer and R. Holzman. Approximations of pseudo-boolean functions: application to game theory. *ZOR - Methods and Models of Operations Research*, 36:3–21, 1992.

207. P.L. Hammer, E.L. Johnson, and U.N. Peled. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179–206, 1975.

208. P.L. Hammer, M.W. Padberg, and U.N. Peled. Constraint pairing in integer programming. *INFOR*, 13:68–81, 1975.

209. P.L. Hammer and U.N. Peled. Computing low capacity 0-1 knapsack polytopes. *ZOR - Methods and Models of Operations Research*, 26:243–149, 1982.

210. P.L. Hammer and D.J. Rader Jr. Efficient methods for solving quadratic 0-1 knapsack problems. *INFOR*, 35:170–182, 1997.

211. S. Hanafi and A. Fréville. An efficient tabu search approach for the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 106:659–675, 1998.

212. S. Hanafi, A. Fréville, and A. El Abdellaoui. Comparison of heuristics for the 0-1 multidimensional knapsack problems. In I.H. Osman and J.P. Kelly, editors, *Meta-Heuristics: Theory & Applications*. Kluwer, 1996.

213. M.P. Hansen. Tabu search for multiobjective combinatorial optimization: TAMOCO. *Control and Cybernetics*, 29:799–818, 2000.

214. D. Hartvigsen and E. Zemel. The complexity of lifted inequalities for the knapsack problem. *Discrete Applied Mathematics*, 39:113–123, 1992.

215. S. Hashizume, M. Fukushima, N. Katoh, and T. Ibaraki. Approximation algorithms for combinatorial fractional programming problems. *Mathematical Programming*, 37:255–267, 1987.

216. C. Haul and S. Voss. Using surrogate constraints in genetic algorithm for solving multi-dimensional knapsack problems. In D.L. Woodruff, editor, *Advances in Computational and Stochastic Optimization, Logic Programming and Heuristic Search*. Kluwer, 1998.

217. M. Held and R.M. Karp. The traveling salesman problem and minimum spanning trees: part II. *Mathematical Programming*, 1:6–25, 1971.

218. C. Helmberg. Semidefinite programming for combinatorial optimization. Technical Report ZIP-Report ZR-00-34, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2000. Habilitation-thesis.

219. C. Helmberg, F. Rendl, and R. Weismantel. Quadratic knapsack relaxations using cutting planes and semidefinite programming. In *Integer Programming and Combinatorial Optimization, 5th IPCO conference*, volume 1084 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 1996.

220. C. Helmberg, F. Rendl, and R. Weismantel. A semidefinite programming approach to the quadratic knapsack problem. *Journal of Combinatorial Optimization*, 4:197–215, 2000.

221. J. Henderson and J.P. Scott. *Securitization*. Woodhead-Faulkner, 1988.

222. M.I. Henig. Risk criteria in a stochastic knapsack problem. *Operations Research*, 38:820–825, 1990.

223. M. Hifi. Exact algorithms for large-scale unconstrained two and three staged unconstrained cutting problems. *Computational Optimization and Applications*, 18:63–88, 2001.

224. M. Hifi and S. Sadfi. The knapsack sharing problem: An exact algorithm. *Journal of Combinatorial Optimization*, 6:35–54, 2002.

225. M. Hifi, S. Sadfi, and A. Sbihi. An efficient algorithm for the knapsack sharing problem. *Computational Optimization and Applications*, 23:27–45, 2002.

226. M. Hifi and V. Zissimopoulos. Constrained two-dimensional cutting: an improvement of Christofides and Whitlock's exact algorithm. *Journal of the Operational Research Society*, 48:324–331, 1997.

227. R.R. Hill and C.H. Reilly. The effects of coefficient correlation structure in two-dimensional knapsack problems on solution performance. *Management Science*, 46:302–317, 2000.

228. F.S. Hillier. Efficient heuristic procedures for integer linear programming with an interior. *Operations Research*, 17:600–637, 1969.

229. D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.

230. D.S. Hirschberg and C.K. Wong. A polynomial-time algorithm for the knapsack problem with two variables. *Journal of the ACM*, 23:147–154, 1976.

231. C.A.R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

232. D. Hochbaum. A nonlinear knapsack problem. *Operations Research Letters*, 17:103–110, 1995.

233. D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

234. J. Hoffstein, J. Pipher, and J.H. Silverman. NTRU: a ring based public key cryptosystem. In *Algorithmic Number Theory (ANTS III)*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.

235. M. Hofri. *Probabilistic Analysis of Algorithms: on Computing Methodologies for Computer Algorithms Performance Evaluation*. Springer, 1987.

236. R.C. Holte. Combinatorial auctions, knapsack problems, and hill-climbing search. In *Advances in Artificial Intelligence (AI-2001)*, volume 2056 of *Lecture Notes in Computer Science*, pages 57–66. Springer, 2001.

237. J.A. Hoogeveen, H. Oosterhout, and S.L. van de Velde. New lower and upper bounds for scheduling around a small common due date. *Operations Research*, 42:102–110, 1994.

238. E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21:277–292, 1974.

239. T.C. Hu and M.L. Lenard. Optimality of a heuristic algorithm for a class of knapsack problems. *Operations Research*, 24:193–196, 1976.

240. M.S. Hung and J.C. Fisk. An algorithm for 0-1 multiple knapsack problems. *Naval Research Logistics Quarterly*, 24:571–579, 1978.

241. O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problem. *Journal of the ACM*, 22:463–468, 1975.

242. O.H. Ibarra and C.E. Kim. Approximation algorithms for certain scheduling problems. *Mathematics of Operations Research*, 4:197–204, 1978.

243. H. Iida. A note on the max-min 0-1 knapsack problem. *Journal of Combinatorial Optimization*, 3:89–94, 1999.

244. H. Iida and T. Uno. A short note on the reducibility of the collapsing knapsack problem. *Journal of the Operations Research Society of Japan*, 45:293–298, 2002.

245. G.P. Ingargiola and J.F. Korsh. Reduction algorithm for zero-one single knapsack problems. *Management Science*, 20:460–463, 1973.

246. G.P. Ingargiola and J.F. Korsh. A general algorithm for one-dimensional knapsack problems. *Operations Research*, 25:752–759, 1977.

247. R.H.F. Jackson, P.T. Boggs, S.G. Nash, and S. Powell. Guidelines for reporting results of computational experiments. Report of the ad hoc committee. *Mathematical Programming*, 49:413–425, 1991.

248. A. Jaszkiewicz. On the performance of multiple objective genetic local search on the 0/1 knapsack problem. A comparative experiment. Technical Report RA–002/2000, Institute of Computing Science, Poznań University of Technology, 2000.

249. L. Jenkins. A bicriteria knapsack program for planning remediation of contaminated lightstation sites. *European Journal of Operational Research*, 140:427–433, 2002.

250. R.G. Jeroslow. Trivial integer programs unsolvable by branch-and-bound. *Mathematical Programming*, 6:105–109, 1974.

251. D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

252. D.S. Johnson and K.A. Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8:1–14, 1983.

253. E.L. Johnson, A. Mehrotra, and G.L. Nemhauser. Min-cut clustering. *Mathematical Programming*, 62:133–152, 1993.

254. E.L. Johnson and M.W. Padberg. A note on the knapsack probem with special ordered sets. *Operations Research Letters*, 1:18–22, 1981.

255. R.E. Johnston and L.R. Khan. A note on dominance in unbounded knapsack problems. *Asia-Pacific Journal of Operational Research*, 12:145–160, 1995.

256. D. Jungnickel. *Graphs, Networks and Algorithms*. Springer, 1999.

257. R. Kannan. A polynomial algorithm for the two-variable integer programming problem. *Journal of the ACM*, 27:118–122, 1980.

258. R.M. Karp. The fast approximate solution of hard combinatorial problems. In *Proceedings of the 6th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 15–31, 1975.

259. M.H. Karwan and R.L. Rardin. Some relationships between Lagrangian and surrogate duality in integer programming. *Mathematical Programming*, 17:320–334, 1979.

260. A.V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

261. C.A. Kaskavelis and M.C. Caramanis. Efficient Lagrangian relaxation algorithms for industry size job-shop scheduling problems. *IIE Transactions*, 30:1085–1097, 1998.

262. H. Kellerer. An FPTAS for the multiperiod knapsack problem. unpublished manuscript.

263. H. Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In *Randomization, Approximation and Combinatorial Optimization (RANDOM-APPROX'99)*, volume 1671 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 1999.

264. H. Kellerer, V. Kotov, M.G. Speranza, and Z. Tuza. Semi on–line algorithms for the partition problem. *Operations Research Letters*, 21:235–242, 1997.

265. H. Kellerer, R. Mansini, and M.G. Speranza. Two linear approximation algorithms for the subset-sum problem. *European Journal of Operational Research*, 120:289–296, 2000.

266. H. Kellerer and U. Pferschy. Improved dynamic programming in connection with an FPTAS for the knapsack problem. to appear in *Journal of Combinatorial Optimization*.

267. H. Kellerer and U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3:59–71, 1999.

268. H. Kellerer, U. Pferschy, R. Mansini, and M.G. Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *Journal of Computer and System Sciences*, 66:349–370, 2003.

269. S. Khan, K.F. Li, E.G. Manning, and M.M. Akbar. Solving the knapsack problem for adaptive multimedia system. *Studia Informatica Universalis*, 2:161–182, 2002.

270. D.G. Kirkpatrick and R. Seidel. The ultimate planer convex hull algorithm? *SIAM Journal on Computing*, 15:287–299, 1986.

271. K.C. Kiwiel. A survey of bundle methods for nondifferentiable optimization. In M. Iri and K. Tanabe, editors, *Mathematical programming and its applications*. KTK Scientific Publishers, 1989.

272. K. Klamroth and M.M. Wiecek. Dynamic programming approaches to the multiple criteria knapsack problem. *Naval Research Logistics*, 47:57–76, 2000.

273. K. Klamroth and M.M. Wiecek. Time-dependent capital budgeting with multiple criteria. In *Research and Practice in Multiple Criteria Decision Making*, volume 487 of *Lecture Notes in Economics and Mathematical Systems*, pages 421–432. Springer, 2000.

274. K. Klamroth and M.M. Wiecek. A time-dependent single-machine scheduling knapsack problem. *European Journal of Operational Research*, 135:17–26, 2001.

275. P. Klein and F. Meyer auf der Heide. A lower bound for the knapsack problem on random access machines. *Acta Informatica*, 19:385–395, 1983.

276. A.J. Kleywegt and J.D. Papastavrou. The dynamic and stochastic knapsack problem. *Operations Research*, 46:17–35, 1998.

277. A.J. Kleywegt and J.D. Papastavrou. The dynamic and stochastic knapsack problem with random sized items. *Operations Research*, 49:26–41, 2001.

278. A.J. Kleywegt, A. Shapiro, and T. Homem-de-Mello. The sample average approximation method for stochastic discrete optimization. *SIAM Journal on Optimization*, 12:479–502, 2001.

279. D.E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*. Addison-Wesley, 1973.

280. G.A. Kochenberger, B.A. McCarl, and F.P. Wyman. A heuristic for general integer programming. *Decision Sciences*, 5:36–44, 1974.

281. R. Kohli and R. Krishnamurti. A total-value greedy heuristic for the integer knapsack problem. *Operations Research Letters*, 12:65–71, 1992.

282. R. Kohli and R. Krishnamurti. Joint performance of greedy heuristics for the integer knapsack problem. *Discrete Applied Mathematics*, 56:37–48, 1995.

283. P.J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13:723–735, 1967.

284. S.G. Kolliopoulos and G. Steiner. Partially ordered knapsack and applications to scheduling. In *European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 612–624. Springer, 2002.

285. R. Korn. *Optimal Portfolios: Stochastic Models for Optimal Investment and Risk Management in Continuous Time*. World Scientific, 1997.

286. B. Korte and R. Schrader. On the existence of fast approximation schemes. In O.L. Mangasarian, R.R. Meyer, and S.M. Robinson, editors, *Nonlinear Programming*, volume 4, pages 415–437. Academic Press, 1981.

287. M. Kostreva, W. Ogryczak, and D.W. Tonkyn. Relocation problems arising in conservation biology. *Computers and Mathematics with Applicatons*, 37:135–150, 1999.

288. A. Kothari, D.C. Parkes, and S. Suhri. Approximately-strategy proof and tractable multi-unit auctions. In *Proceedings of the 4th ACM Conference on Electronic Commerce (EC-03)*, 2003.

289. M.Y. Kovalyov. A rounding technique to construct approximation algorithms for knapsack and partition-type problems. *Applied Mathematics and Computer Science*, 6:789–801, 1996.

290. T. Kuno, H. Konno, and E. Zemel. A linear-time algorithm for solving continuous maximin knapsack problems. *Operations Research Letters*, 10:23–26, 1991.

291. W. Kwak, Y. Shi, H. Lee, and C.F. Lee. Capital budgeting with multiple criteria and multiple decision makers. *Review of Quantitative Finance and Accounting*, 7:97–112, 1996.

292. J.C. Lagarias. Knapsack public key cryptosystems and diophantine approximation. In *Advances in Cryptology – CRYPTO '83*, pages 3–23. Plenum Press, 1984.

293. J.C. Lagarias and A.M. Odlyzko. Solving low-density subset sum problems. *Journal of the ACM*, 32:229–246, 1985.

294. M. Laumanns, E. Zitzler, and L. Thiele. On the effect of archiving, elitism, and density based selection in evolutionary multi-objective optimization. In *First International Conference on Evolutionary Multi-Criterion Optimization*, volume 1993 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2001.

295. E.L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4:339–356, 1979.

296. J.S. Lee and M. Guignard. An approximate algorithm for multidimensional zero-one knapsack problems– a parametric approach. *Management Science*, 34:402–410, 1988.

297. A.K. Lenstra, H.W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

298. H.W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

299. K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce (EC-00)*, pages 66–76, 2000.

300. K. Leyton-Brown, Y. Shoham, and M. Tennenholtz. An algorithm for multi-unit combinatorial auctions. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 56–61, 2000.

301. E. Lin. A bibliographical survey on some well-known non-standard knapsack problems. *INFOR*, 36:274–317, 1998.

302. Y. Liu. The fully polynomial approximation algorithm for the 0-1 knapsack problem. *Theory of Computing Systems*, 35:559–564, 2002.

303. A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: a survey. *European Journal of Operational Research*, 141:241–252, 2002.

304. A. Lodi, S. Martello, and D. Vigo. Approximation algorithms for the two–dimensional oriented bin packing problem. *European Journal of Operational Research*, 112:158–166, 1999.

305. A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123:379–396, 2002.

306. J.H. Lorie and L.J. Savage. Three problems in capital rationing. *The Journal of Business*, 28:229–239, 1955.

307. R. Loulou and E. Michaelides. New greedy-like heuristics for the multidimensional 0-1 knapsack problem. *Operations Research*, 27:1101–1114, 1979.

308. L. Lovász and A. Schrijver. Cones of matrices and set-functions and 0-1 optimization. *SIAM Journal on Optimization*, 1:166–190, 1991.

309. G.S. Lueker. Two NP-complete problems in nonnegative integer programming. Technical Report 178, Computer Science Laboratory, Princeton University, 1975.

310. G.S. Lueker. On the average difference between the solutions to linear and integer knapsack problems. In *Applied Probability - Computer Science, the Interface*, volume 1, pages 489–504. Birkhäuser, 1982.

311. G.S. Lueker. Average-case analysis of off-line and on-line knapsack problems. *Journal of Algorithms*, 29:277–305, 1998.

312. M.J. Magazine and M.-S. Chern. A note on approximation schemes for multidimensional knapsack problems. *Mathematics of Operations Research*, 9:244–247, 1984.

313. M.J. Magazine, G.L. Nemhauser, and L.E. Trotter. When a greedy solution solves a class of knapsack problems. *Operations Research*, 23:207–217, 1975.

314. M.J. Magazine and O. Oguz. A fully polynomial approximation algorithm for the 0–1 knapsack problem. *European Journal of Operational Research*, 8:270–273, 1981.

315. M.J. Magazine and O. Oguz. A heuristic algorithm for the multidimensional zero-one knapsack problem. *European Journal of Operational Research*, 16:319–326, 1984.

316. J.W. Mamer and K.E. Schilling. On the growth of random knapsack. *Discrete Applied Mathematics*, 28:223–230, 1990.

317. R. Mansini and U. Pferschy. Securitization of financial assets: approximation in theory and practice. Technical Report 03/2001, Faculty of Economics, University of Graz, 2001.

318. R. Mansini and M.G. Speranza. On selecting a portfolio of lease contracts in an asset-backed securitization process. In C. Zopounidis, editor, *New Operational Approaches for Financial Modelling*. Physica, 1997.

319. A. Marchetti-Spaccamela and C. Vercellis. Stochastic on-line knapsack problems. *Mathematical Programming*, 68:73–104, 1995.

320. D.W. Marquardt. An algorithm for least squares estimation of nonlinear parameters. *SIAM Journal on Applied Mathematics*, 11:431–441, 1963.

321. S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45:414–424, 1999.

322. S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123:325–332, 2000.

323. S. Martello and P. Toth. Branch and bound algorithms for the solution of general unidimensional knapsack problems. In M. Roubens, editor, *Advances in Operations Research*, pages 295–301. North-Holland, 1977.

324. S. Martello and P. Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1:169–175, 1977.

325. S. Martello and P. Toth. A note on the Ingargiola-Korsh algorithm for one-dimensional knapsack problems. *Operations Research*, 28:1226–1227, 1980.

326. S. Martello and P. Toth. Optimal and canonical solutions of the change-making problem. *European Journal of Operational Research*, 4:322–329, 1980.

327. S. Martello and P. Toth. Solution of the zero-one multiple knapsack problem. *European Journal of Operational Research*, 4:276–283, 1980.

328. S. Martello and P. Toth. A bound and bound algorithm for the zero-one multiple knapsack problem. *Discrete Applied Mathematics*, 3:275–288, 1981.

329. S. Martello and P. Toth. Heuristic algorithms for the multiple knapsack problem. *Computing*, 27:93–112, 1981.

330. S. Martello and P. Toth. A mixture of dynamic programming and branch-and-bound for the subset-sum problem. *Management Science*, 30:765–771, 1984.

331. S. Martello and P. Toth. Worst-case analysis of greedy algorithms for the subset-sum problem. *Mathematical Programming*, 28:198–205, 1984.

332. S. Martello and P. Toth. Approximation schemes for the subset-sum problem: survey and experimental results. *European Journal of Operational Research*, 22:56–69, 1985.

333. S. Martello and P. Toth. A new algorithm for the 0-1 knapsack problem. *Management Science*, 34:633–644, 1988.

334. S. Martello and P. Toth. An exact algorithm for large unbounded knapsack problems. *Operations Research Letters*, 9:15–20, 1990.

335. S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. J. Wiley, 1990.

336. S. Martello and P. Toth. Upper bounds and algorithms for hard 0-1 knapsack problems. *Operations Research*, 45:768–778, 1997.

337. M. Mastrolilli and M. Hutter. Hybrid rounding techniques for knapsack problems. to appear in *Discrete Applied Mathematics*, 2003.

338. G.B. Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 28:486–490, 1897.

339. K. Mathur, H.M. Salkin, and B.B. Mohanty. A note on a general non-linear knapsack problem. *Operations Research Letters*, 5:79–81, 1986.

340. M. Meanti, A.H.G. Rinnooy Kan, L. Stougie, and C. Vercellis. A probabilistic analysis of the multiknapsack value function. *Mathematical Programming*, 46:237–247, 1990.

341. N. Megiddo. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research*, 4:414–424, 1979.

342. N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of the ACM*, 30:852–865, 1983.

343. N. Megiddo and A. Tamir. Linear time algorithms for some separable quadratic programming problems. *Operations Research Letters*, 13:203–211, 1993.

344. P. Mejia-Alvarez, E.V. Levner, and D. Mosse. An integrated heuristic approach to power-aware real-time scheduling. In *International Workshop on Power Aware Computer Systems (PACS'02)*, number 2325 in Lecture Notes on Computer Science. Springer, 2002.

345. R. Merkle and M. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, IT-24:525–530, 1978.

346. F. Meyer auf der Heide. A polynomial linear search algorithm for the n-dimensional knapsack problem. *Journal of the ACM*, 31:668–676, 1984.

347. P. Michelon and L. Veilleux. Lagrangean methods for the 0-1 quadratic knapsack problem. *European Journal of Operational Research*, 92:326–341, 1996.

348. G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:1–4, 1964.

349. R. Morabito and V. Garcia. The cutting stock problem in a hardboard industry: a case study. *Computers and Operations Research*, 25:469–485, 1998.

350. T.L. Morin and R.E. Marsten. A hybrid approach to discrete applied mathematics. *Mathematical Programming*, 14:21–40, 1978.

351. H. Morita, H. Ishii, and T. Nishida. Stochastic linear knapsack programming problem and its application to a portfolio selection problem. *European Journal of Operational Research*, 40:329–336, 1989.

352. D.P. Morton and R.K. Wood. On a stochastic knapsack problem and generalizations. In D.L. Woodruff, editor, *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search*, pages 149–169. Kluwer, 1998.

353. M. Moser, D.P. Jokanovic, and N. Shiratori. An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE Transactions A*, E80:582–589, 1997.

354. H. Müller-Merbach. Improved upper bound for the zero-one knapsack problem. A note on the paper by Martello and Toth. *European Journal of Operational Research*, 2:212–213, 1979.

355. R.M. Nauss. The 0-1 knapsack problem with multiple choice constraint. *European Journal of Operational Research*, 2:125–131, 1978.

356. R.M. Nauss. *Parametric Integer Programming*. PhD thesis, University of Missouri, 1979.

357. A. Neebe and D. Dannenbring. Algorithms for a specialized segregated storage problem. Technical Report 77-5, University of North Carolina, 1977.

358. G.L. Nemhauser and Z. Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15:494–505, 1969.

359. G.L. Nemhauser and P.H. Vance. Lifted cover facets of the 0-1 knapsack polytope with GUB constraints. *Operations Research Letters*, 16:255–263, 1994.

360. G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. J. Wiley, 1988.

361. J.J. Norton and P.R. Spellman, editors. *Asset Securitization: International Financial and Legal Perspectives*. Blackwell, 1991.

362. P.J. O'Rourke. *Holidays in Hell*. Grove Press, 2000.

363. M.A. Osorio, F. Glover, and P.L. Hammer. Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions. Technical Report HCES-08-00, Hearin Center for Enterprise Science, University of Mississippi, 2000.

364. M. Ozden. A solution procedure for general knapsack problems with a few constraints. *Computers and Operations Research*, 15:145–155, 1988.

365. M.W. Padberg. A note on zero-one programming. *Operations Research*, 23:833–837, 1975.

366. M.W. Padberg. $(1,k)$ configurations and facets for packing problems. *Mathematical Programming*, 18:94–99, 1980.

367. C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

368. C.H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 86–92, 2000.

369. J.D. Papastavrou, S. Rajagopalan, and A.J. Kleywegt. The dynamic and stochastic knapsack problem with deadlines. *Management Science*, 42:1706–1718, 1996.

370. K. Park, K. Lee, and S. Park. An extended formulation approach to the edge-weighted maximal clique problem. *European Journal of Operational Research*, 95:671–682, 1996.

371. R. Parra-Hernandez and N. Dimopoulos. A new heuristic for solving the multi-choice multidimensional knapsack problem. Technical report, Department of Electrical and Computer Engineering, University of Victoria, 2002.

372. C.C. Peterson. Computational experience with variants of the Balas algorithm applied to the selection of R&D projects. *Management Science*, 13:736–750, 1967.

373. C.C. Peterson. A capital budgeting heuristic algorithm using exchange operations. *AIIE Transactions*, 6:143–150, 1974.

374. U. Pferschy. Dynamic programming revisited: improving knapsack algorithms. *Computing*, 63:419–430, 1999.

375. U. Pferschy. Stochastic analysis of greedy algorithms for the subset sum problem. *Central European Journal of Operations Research*, 7:53–70, 1999.

376. U. Pferschy, D. Pisinger, and G.J. Woeginger. Simple but efficient approaches for the collapsing knapsack problem. *Discrete Applied Mathematics*, 77:271–280, 1997.

377. J.C. Picard and H.D. Ratliff. Minimum cuts and related problems. *Networks*, 5:357–370, 1974.

378. H. Pirkul. A heuristic solution procedure for the multiconstraint zero–one knapsack problem. *Naval Research Logistics*, 34:161–172, 1987.

379. H. Pirkul and S. Narasimhan. Efficient algorithms for the multiconstraint general knapsack problem. *IIE Transactions*, 18:195–203, 1986.

380. D. Pisinger. Dominance relations in unbounded knapsack problems. Technical Report 94/33, DIKU, University of Copenhagen, 1994.

381. D. Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem. *European Journal of Operational Research*, 87:175–187, 1995.

382. D. Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83:394–410, 1995.

383. D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45:758–767, 1997.

384. D. Pisinger. A fast algorithm for strongly correlated knapsack problems. *Discrete Applied Mathematics*, 89:197–212, 1998.

385. D. Pisinger. Core problems in knapsack algorithms. *Operations Research*, 47:570–575, 1999.

386. D. Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114:528–541, 1999.

387. D. Pisinger. Linear time algorithms for knapsack problems with bounded weights. *Journal of Algorithms*, 33:1–14, 1999.

388. D. Pisinger. A minimal algorithm for the bounded knapsack problem. *INFORMS Journal on Computing*, 34:75–84, 2000.

389. D. Pisinger. Budgeting with bounded multiple-choice constraints. *European Journal of Operational Research*, 129:471–480, 2001.

390. D. Pisinger. Heuristics for the container loading problem. *European Journal of Operational Research*, 141:382–392, 2002.

391. D. Pisinger. The quadratic knapsack problem—a survey. submitted, 2002.

392. D. Pisinger. Dynamic programming on the word RAM. *Algorithmica*, 35:128–145, 2003.

393. D. Pisinger. Where are the hard knapsack problems? Technical Report 03/08, DIKU, University of Copenhagen, 2003.

394. D. Pisinger and P. Toth. Knapsack problems. In D.Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 1, pages 299–428. Kluwer, 1998.

395. G. Plateau and M. Elkihel. A hybrid method for the 0-1 knapsack problem. *Methods of Operations Research*, 49:277–293, 1985.

396. V. Poirriez, N. Yanev, and R. Andonov. Towards reduction of a class of intractable unbounded knapsack problem. Technical report, University of Valenciennes, 2002.

397. M.E. Posner and M. Guignard. The collapsing 0-1 knapsack problem. *Mathematical Programming*, 15:155–161, 1978.

398. D.J. Rader Jr. and G.J. Woeginger. The quadratic 0-1 knapsack problem with series-parallel support. *Operations Research Letters*, 30:159–166, 2002.

399. T. Radzik. Newton's method for fractional combinatorial optimization. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS'92)*, pages 659–669, 1992.

400. T. Radzik. Fractional combinatorial optimization. In D.Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization, Volume 1*, pages 429–478. Kluwer, 1998.

401. G.R. Raidl. Weight-codings in a genetic algorithm for the multiconstraint knapsack problem. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation*, pages 596–603, 1999.

402. B. Ram and S. Sarin. An algorithm for the 0-1 equality knapack problem. *Journal of the Operational Research Society*, 39:1045–1049, 1988.

403. A.B. Rasmussen and R. Sandvik. Kvaliteten af grænseværdier for det kvadratiske knapsack problem, 2003. Project 02-09-7, DIKU, University of Copenhagen (D. Pisinger, supervisor).

404. S.J. Rassenti, V.L. Smith, and R.L. Bulfin. A combinatorial auction mechanism for airport time slot allocation. *Bell Journal of Economics*, 13:402–417, 1982.

405. B. Render and R.M. Stair. *Quantitative Analysis for Management*. Prentice Hall, 8th edition, 2003.

406. J. Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17:200–207, 1970.

407. A.H.G. Rinnooy Kan, L. Stougie, and C. Vercellis. A class of generalized greedy algorithms for the multi-knapsack problem. *Discrete Applied Mathematics*, 42:279–290, 1993.

408. R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1976.

409. M.J. Rosenblatt and Z. Sinuany-Stern. Generating the discrete efficient frontier to the capital budgeting problem. *Operations Research*, 37:384–394, 1989.

410. K.W. Ross and D.H.K. Tsang. The stochastic knapsack problem. *IEEE Transactions on Communications*, 37:740–747, 1989.

411. S.M. Ross. *Stochastic Processes*. J. Wiley, 2nd edition, 1995.

412. S.M. Ross. *Introduction to Probability Models*. Academic Press, 8th edition, 2003.

413. H.M. Safer and J.B. Orlin. Fast approximation schemes for multi-criteria combinatorial optimization. Technical Report 3756–95, MIT Sloan School of Management, 1995.

414. H.M. Safer and J.B. Orlin. Fast approximation schemes for multi-criteria flow, knapsack, and scheduling problems. Technical Report 3757–95, MIT Sloan School of Management, 1995.

415. S. Sahni. Approximate algorithms for the 0–1 knapsack problem. *Journal of the ACM*, 22:115–124, 1975.

416. A. Salomaa. *Public-Key Cryptography*. Springer, 1996.

417. N. Samphaiboon and T. Yamada. Heuristic and exact algorithms for the precedence-constrained knapsack problem. *Journal of Optimization Theory and their Applications*, 105:659–676, 2000.

418. H.E. Scarf. Production sets with indivisibilities– part II: the case of two activities. *Econometrica*, 49:395–423, 1981.

419. H. Schachnai and T. Tamir. Noah Bagels – some combinatorial aspects. In *International Conference on FUN with Algorithms (FUN)*, 1998.

420. H. Schachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, 29:442–467, 2001.

421. H. Schachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. *Journal of Scheduling*, 4:313–338, 2001.

422. H. Schachnai and T. Tamir. Tight bounds for on-line class-constrained packing. In *Latin American Symposium on Theoretical Informatics (LATIN)*, volume 2286 of *Lecture Notes in Computer Science*. Springer, 2002.

423. J.D. Schaffer. Multiple objective optimization with vector evalutated genetic algorithms. In *Genetic Algorithms and their Applications: Proceedings of the First International Conference on Genetic Algorithms*, pages 93–100, 1985.

424. K.E. Schilling. The growth of m-constraint random knapsacks. *European Journal of Operational Research*, 46:109–112, 1990.

425. K.E. Schilling. Random knapsacks with many constraints. *Discrete Applied Mathematics*, 48:163–174, 1994.

426. C.P. Schnorr. A hierarchy of polynomial time lattice base reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987.

427. C.P. Schnorr. A more efficient algorithm for lattice base reduction. *Journal of Algorithms*, 9:47–62, 1988.

428. A. Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency*. Springer, 2003.

429. S. Senju and Y. Toyoda. An approach to linear programming with 0–1 variables. *Management Science*, 15:196–207, 1968.

430. A. Shamir. A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem. *IEEE Transactions on Information Theory*, IT-30:699–704, 1984.

431. J.F. Shapiro. Dynamic programming algorithms for the integer programming problem - I: the integer programming problem viewed as a knapsack type problem. *Operations Research*, 16:103–121, 1968.

432. D.X. Shaw and G. Cho. The critical item, upper bounds and a branch-and-bound algorithm for the tree knapsack problem. *Networks*, 31:205–216, 1998.

433. D.X. Shaw, G. Cho, and H. Chang. A depth-first dynamic programming procedure for the extended tree knapsack problem in local access network design. *Telecommunication Systems*, 7:29–43, 1997.

434. W. Shih. A branch and bound method for the multiconstraint zero-one knapsack problem. *Journal of the Operational Research Society*, 30:369–378, 1979.

435. D. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming A*, 62:461–474, 1993.

436. A. Sinha and A.A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27:503–515, 1979.

437. S.S. Skiena. Who is interested in algorithms and why? - lessons from the Stony Brook algorithms repository. In *Second Workshop on Algorithm Engineering (WAE'98)*, pages 204–212, Saarbrücken, Germany, 1998.

438. M. Sniedovich. Preference order stochastic knapsack problems: methodological issues. *Journal of the Operational Research Society*, 31:1025–1032, 1980.

439. M. Sniedovich. Some comments on preference order dynamic programming models. *Journal of Mathematical Analysis and Applications*, 79:489–501, 1981.

440. J. Soeiro Ferreira, M.A. Neves, and P.F. Castro. A two-phase roll cutting problem. *European Journal of Operational Research*, 44:185–196, 1990.

441. N.Y. Soma and P. Toth. An exact algorithm for the subset sum problem. *European Journal of Operational Research*, 136:57–66, 2002.

442. N.Y. Soma, A.S.I. Zinober, H.H. Yanasse, and P.J. Harley. A polynomial approximation scheme for the subset sum problem. *Discrete Applied Mathematics*, 57:243–253, 1995.

443. A.L. Soyster, B. Lev, and W. Slivka. Zero-one programming with many variables nad few constraints. *European Journal of Operational Research*, 2:195–201, 1978.

444. K.E. Stecke and I. Kim. A study of part type selection approaches for short-term production planning. *International Journal of Flexible Manufacturing Systems*, 1:7–29, 1988.

445. E. Steinberg and M.S. Parks. A preference order dynamic program for a knapsack problem with stochastic rewards. *Journal of the Operational Research Society*, 30:141–147, 1979.

446. J.F. Sturm. Using SeDuMi 1.02, a Matlab toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11-12:625–653, 1999.

447. J.F. Sturm. SeDuMi 1.05, Matlab toolbox for solving optimization problems over symmetric cones, 2002. http://fewcal.kub.nl/sturm/software/sedumi.html.

448. H. Süral, L.N. van Wassenhove, and C.N. Potts. The bounded knapsack problem with setups. Technical Report 97/71/TM, INSEAD, 1997.

449. K. Szkatula. The growth of multi-constraint random knapsacks with various right-hand sides of the constraints. *European Journal of Operational Research*, 73:199–204, 1994.

450. K. Szkatula. The growth of multi-constraint random knapsacks with large right-hand sides of the constraints. *Operations Research Letters*, 21:25–30, 1997.

451. K. Szkatula and M. Libura. Probabilistic analysis of simple algorithms for binary knapsack problem. *Control and Cybernetics*, 12:147–157, 1983.

452. K. Szkatula and M. Libura. On probabilistic properties of greedy-like algorithms for the binary knapsack problem. Technical Report 154, Polska Akademia Nauk, Instytut Badan Systemowych, Warsaw, 1987.

453. J. Teghem, D. Tuyttens, and E.L. Ulungu. An interactive heuristic method for multi-objective combinatorial optimization. *Computers and Operations Research*, 27:621–634, 2000.

454. J.-Y. Teng and G.-H. Tzeng. A multiobjective planning approach for selecting non-independent transportation investment alternatives. *Transportation Research B*, 30:291–307, 1996.

455. A. Thesen. Scheduling of computer programs in a multiprogramming environment. *BIT*, 13:208–216, 1973.

456. A. Thesen. A recursive branch and bound algorithm for the multidimensional knapsack problem. *Naval Research Logistics Quarterly*, 22:341–353, 1975.

457. B. Thiongane, A. Nagih, and G. Plateau. Lagrangean heuristics combined with reoptimization for the 0-1 biknapsack problem. Computer Science Laboratory, University of Paris-Nord, 2003.

458. B.N. Tien and T.C. Hu. Error bounds and the applicability of the greedy solution to the coin-changing problem. *Operations Research*, 25:404–418, 1977.

459. G. Tinhofer and H. Schreck. The bounded subset sum problem is almost everywhere randomly decidable in $O(n)$. *Information Processing Letters*, 23:11–17, 1986.

460. K.C. Toh, R.H. Tutuncu, and M.J. Todd. SDPT3 – a MATLAB software for semidefinite-quadratic-linear programming, 2002. http://www.math.nus.edu.sg/~mattohkc/ sdpt3.html.

461. Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science*, 21:1417–1427, 1975.

462. E.L. Ulungu and J. Teghem. Solving multi-objective knapsack problems by a branch-and-bound procedure. In J. Climacao, editor, *Multicriteria Analysis*, pages 269–278. Springer, 1997.

463. E.L. Ulungu, J. Teghem, P.H. Fortems, and D. Tuyttens. MOSA method: a tool for solving multiobjective combinatorial optimization problems. *Journal of Multi-Criteria Decision Analysis*, 8:221–236, 1999.

464. J.M. Valério de Carvalho and A.J. Guimarães Rodrigues. A computer based interactive approach to a two-stage cutting stock problem. *INFOR*, 32:243–252, 1994.

465. J.M. Valério de Carvalho and A.J. Guimarães Rodrigues. An LP-based approach to a two-stage cutting stock problem. *European Journal of Operational Research*, 84:580–589, 1995.

466. S. van de Geer and L. Stougie. On rates of convergence and asymptotic normality in the multiknapsack problem. *Mathematical Programming*, 51:349–358, 1991.

467. R.L.M.J. van de Leensel, C.P.M. van Hoesel, and J.J. van de Klundert. Lifting valid inequalities for the precedence constrained knapsack problem. *Mathematical Programming A*, 86:161–185, 1999.

468. S.L. van de Velde and J.M. Worm. Multi-period planning of road maintenance: a multiple-choice multi-knapsack problem. Technical Report LPOM-94-6, Department of Mechanical Engineering, University of Twente, 1994.

469. R. van Slyke and Y. Young. Finite horizon stochastic knapsacks with applications to yield management. *Operations Research*, 48:155–172, 2000.

470. P.H. Vance, C. Barnhart, E.L. Johnson, and G.L. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3:111–130, 1994.

471. M. Vasquez and J.-K. Hao. A hybrid approach for the 0–1 multidimensional knapsack problem. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 328–333, 2001.

472. S. Vaudenay. Cryptanalyis of the Chor-Rivest cryptosystem. In *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 1998.

473. W. Vickrey. Counterspeculations, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

474. M. Visée, J. Teghem, M. Pirlot, and E.L. Ulungu. Two-phases method and branch and bound to solve the bi-objective knapsack problem. *Journal of Global Optimization*, 12:139–155, 1998.

475. B. Vizvari. On pleasant but not completely pleasant knapsack problems. *Annales Universitatis Scientiarum Budapestinensis, Sectio Computatorica*, 19:17–25, 2000.

476. A. Volgenant and S. Marsman. A core approach to the 0-1 equality knapsack problem. *Journal of the Operational Research Society*, 49:86–92, 1998.

477. A. Volgenant and J.A. Zoon. An improved heuristic for the multidimensional 0–1 knapsack problem. *Journal of the Operational Research Society*, 41:963–970, 1990.

478. H.M. Weingartner. Capital budgeting of interrelated projects: survey and synthesis. *Management Science*, 12:485–516, 1966.

479. H.M. Weingartner and D.N. Ness. Methods for the solution of the multidimensional 0/1 knapsack problem. *Operations Research*, 15:83–103, 1967.

480. R. Weismantel. Knapsack problems, test sets and polyhedra. TU Berlin, 1995. Habilitation-thesis.

481. R. Weismantel. On the 0/1 knapsack polytope. *Mathematical Programming*, 77:49–68, 1997.

482. D.J. White. A complementary greedy heuristics for the knapsack problem. *European Journal of Operational Research*, 62:85–95, 1992.

483. G. Wirsching, 1998. personal communication.

484. C. Witzgall. Mathematical methods of site selection for electronic message systems (EMS). Technical report, Applied Mathematics Division, National Bureau of Standards, 1975.

485. C. Witzgall. On one-row linear programs. Technical report, Applied Mathematics Division, National Bureau of Standards, 1977.

486. G.J. Woeginger. On the approximability of average completion time scheduling under precedence constraints. In *28th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *Lecture Notes in Computer Science*, pages 887–897. Springer, 2001.

487. H. Wolkowicz, R. Saigal, and L. Vandenberghe, editors. *Handbook of Semidefinite Programming*, volume 27 of *International Series in Operations Research and Management Science*. Kluwer, 2000.

488. L.A. Wolsey. Facets of linear inequalities in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.

489. L.A. Wolsey. *Integer Programming*. J. Wiley, 1998.

490. T. Yamada and M. Futakawa. Heuristic and reduction algorithms for the knapsack sharing problem. *Computers and Operations Research*, 24:961–967, 1997.

491. T. Yamada, M. Futakawa, and S. Kataoka. Some exact algorithms for the knapsack sharing problem. *European Journal of Operational Research*, 106:177–183, 1998.

492. H.H. Yanasse and N.Y. Soma. An exact pseudopolynomial algorithm for the value independent knapsack problem. In *Proceedings of the XX SBPO, Volume 1, San Salvador*, pages 710–719, 1987.

493. M.-H. Yang. An efficient algorithm to allocate shelf space. *European Journal of Operational Research*, 131:107–118, 2001.

494. G. Yu. On the max-min 0-1 knapsack problem with robust optimization applications. *Operations Research*, 44:407–415, 1996.

495. E. Zak and E. Dereksdottir. A modified lexicographic algorithm for multi-constraint knapsack problems. Technical Report OR-003/0644, Majiq Systems and Software Inc., Redmond, WA, 2001.

496. S.H. Zanakis. Heuristic 0-1 linear programming: an experimental comparison of three methods. *Management Science*, 24:91–104, 1977.

497. E. Zemel. The linear multiple choice knapsack problem. *Operations Research*, 28:1412–1423, 1980.

498. E. Zemel. An $O(n)$ algorithm for the linear multiple choice knapsack problem and related problems. *Information Processing Letters*, 18:123–128, 1984.

499. E. Zemel. Easily computable facets of the knapsack polytope. *Mathematics of Operations Research*, 14:760–764, 1989.

500. N. Zhu and K. Broughan. On dominated terms in the general knapsack problems. *Operations Research Letters*, 21:31–37, 1997.

501. E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, ETH Zürich, 1999.

502. E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3:257–271, 1999.

503. M. Zukerman, L. Jia, T. Neame, and G.J. Woeginger. A polynomially solvable special case of the unbounded knapsack problem. *Operations Research Letters*, 29:13–16, 2001.

Author Index

Abdelaziz, F.B. 402
Adams, W.P. 358
Adleman, L.M. 472, 477
Agrawal, M. 77
Ahrens, J.H. 81
Aittoniemi, L. 203
Akbar, M.M. 282
Allender, E. 77
Alvarez-Valdes, R. 450
Alves, M.J. 401
Andonov, R. 216, 222, 223, 224, 227, 229, 230, 250, 251, 252
Archer, A. 480
Ausiello, G. 40
Averbakh, I. 442

Babayev, D.A. 214, 216
Balas, E. 43, 69, 70, 71, 86, 121, 122, 140, 142, 143, 144, 145, 262, 264, 265, 266, 358, 429, 459
Balev, S. 250, 251, 252
Bank, B. 419
Barcia, P. 118, 119, 318
Barnhart, C. 432, 433, 455
Barr, R.S. 256
Battiti, R. 268
Bauvin, M. 372
Beasley, J.E. 237, 268
Beier, R. 431, 436, 437
Beinsen, L. 462
Bellman, R.E. 20, 21, 75, 131
Ben-Or, M. 118
Bertsimas, D. 250, 261, 266, 267, 268
Bhaskar, K. 390
Billionnet, A. 356, 362, 364, 367, 374, 375, 379, 382, 423
Blum, L. 118
Blum, M. 43
Boggs, P.T. 256
Borgwardt, K.H. 436, 439
Borodin, A. 443
Boyd, E.A. 408
Bradley, S.P. 186
Brandstädt, A. 408
Bretthauer, K.M. 350, 409, 411
Brickell, E.F. 473, 475, 477, 478
Brimkov, V.E. 118
Broughan, K. 216
Brown, J.R. 342
Brzank, J. 436
Bulfin, R.L. 203, 479
Burkard, R.E. 421, 422, 423

Cabot, A.V. 230, 246
Caesar, G.J. 256
Calmels, F. 356, 362, 374, 379, 382
Calvin, J.M. 435, 436
Caprara, A. 74, 162, 166, 254, 255, 272, 273, 274, 277, 278, 279, 301, 302, 303, 304, 305, 309, 351, 356, 358, 359, 360, 362, 373, 374, 375, 378, 379, 382, 384, 450
Captivo, M.E. 393
Caramanis, M.C. 63
Carraresi, P. 362
Carraway, R.L. 433
Carstensen, P.J. 420
Castro, P.F. 452
Cerdeira, J.O. 118, 119
Ceria, S. 358
Chaillou, P. 352, 353, 374, 382
Chajakis, E.D. 190
Chakravarti, N. 112
Chandra, A.K. 254, 339
Chang, H. 402
Chang, S.K. 416
Chaouachi, J. 402
Chekuri, C. 300, 304, 311, 313, 315, 316
Chern, M.-S. 254

Cho, G. 402, 406, 408
Chor, B. 478
Christofides, N. 287, 292, 452
Chu, P.C. 237, 268
Chung, K.L. 425
Chvátal, V. 91, 112, 260
Clarke, M.R.B. 254, 397, 429, 441
Climacao, J. 393, 401
Coello Coello, C.A. 402
Coffman, E.G. 426
Cohn, A.M. 432, 433
Cook, S. 483, 487
Cormen, T.H. 13, 20, 43, 60, 77, 149, 353, 377, 426, 483, 488, VII
Cornuéjols, G. 358
Crama, Y. 240, 242
Crescenzi, P. 40, 381, 483
Csirik, J. 302
Cucker, F. 118
Czyzak, P. 401

Dannenbring, D. 292
Dantchev, S.S. 118
Dantzig, G.B. 120
d'Atri, G. 437, 438, 439
Datta, S. 77
Dawande, M. 315
de Farias, I.R. 272
De Leone, R. 318
de Vries, S. 479
Dembo, R.S. 126, 193, 291, 327
Demir, R. 250, 261, 266, 267, 268
Dereksdottir, E. 273
di Rende, A. 438
Dietrich, B.L. 74
Dijkhuizen, G. 351
Dimopoulos, N. 282, 283
Dinkelbach, W. 422
Diubin, G. 435
Dobkin, D. 118
Dobson, G. 254, 257, 260
Dor, D. 43, 99
Drexl, A. 268
Dudziński, K. 121, 216, 217, 328, 329, 331, 424
Dupacova, J. 433
Dyckhoff, H. 186, 236
Dyer, M.E. 243, 283, 322, 326, 327, 328, 332, 338, 440, 441

Eben-Chaime, M. 392
Eilon, S. 287
El Abdellaoui, A. 268
Elkihel, M. 88
Elton, E.J. 463
El-Yaniv, R. 443
Epstein, J.B. 419
Epstein, P.G. 419
Erlebach, T. 395, 396, 397, 398, 399
Escudero, L.F. 70, 74
Everett, H. 239, 461

Faaland, B.H. 424
Faigle, U. 351
Fayard, D. 121, 144, 145, 245, 419, 452
Faye, A. 364, 367, 375
Feldman, I. 222, 223
Feller, W. 425
Fernandez de la Vega, W. 306, 314
Ferreira, C.E. 327
Feuerman, M. 4
Figueira, J. 393
Finke, G. 81
Finke, U. 186
Fischetti, M. 96
Fisher, M.L. 239, 318, 362
Fisk, J.C. 289, 292
Floyd, R.W. 43
Fontanari, J.F. 442
Fortems, P.H. 401
Fournier, H. 118
Fox, G.E. 256, 257, 258
Fredman, M.L. 60
Fréville, A. 243, 245, 250, 251, 252, 265, 266, 268, 270, 401
Frieze, A.M. 254, 397, 429, 434, 440, 441
Fujisawa, K. 369
Fukushima, M. 423
Futakawa, M. 343, 347

Gallo, G. 349, 355, 374, 379, 382, 384
Gambosi, G. 40
Gandibleux, X. 401
Garcia, V. 450
Garey, M.R. 74, 252, 253, 298, 302, 403, 411, 479, 483, 488
Garín, A. 70
Gavish, B. 238, 242, 243, 244, 245, 247, 270, 271
Gens, G.V. 97, 98, 112, 252, 338, 413

Geoffrion, A.M. 355
Ghosh, D. 112
Gill, A. 416
Gilmore, P.C. 186, 214, 215, 216, 230, 248, 272, 273, 452, 455
Glover, F. 214, 239, 240, 244, 245, 246, 268, 380
Goel, A. 431
Goemans, M.X. 372
Goldberg, A.V. 144, 428, 429, 430, 440
Golden, B.L. 256
Goldschmidt, O. 424
Golubchik, L. 316
Gomory, R.E. 186, 214, 215, 216, 230, 248, 272, 273, 452, 455
Gonen, R. 480
Gottlieb, J. 268
Greenberg, H.J. 203, 213, 214, 216, 222, 223, 240, 244, 419
Grigoriadis, M. 355
Grötschel, M. 369
Gruber, M.J. 463
Guéret, C. 73
Guignard, M. 190, 264, 265, 416, 419
Guimarães Rodrigues, A.J. 452
Güntzer, M.M. 413, 464, 465

Hadjiconstantinou, E. 452
Hagerup, T. 60, 62
Hammer, P.L. 69, 71, 121, 126, 193, 245, 246, 291, 327, 349, 355, 373, 374, 379, 382, 384
Hanafi, S. 268
Hansen, M.P. 401
Hansen, P. 352, 353, 374, 382
Hao, J.-K. 268
Harley, P.J. 96
Hartvigsen, D. 71
Hashizume, S. 423
Haul, C. 268
Hax, A.C. 186
Hegerich, R.L. 203
Held, M. 384
Hellman, M. 473, 474
Helmberg, C. 351, 358, 367, 369, 370, 372, 374
Henderson, J. 466
Henig, M.I. 433
Hifi, M. 344, 347, 452
Hill, R.R. 271
Hillier, F.S. 261, 262, 264
Hirschberg, D.S. 46, 211, 254, 339
Hoare, C.A.R. 142
Hochbaum, D. 410
Hoffstein, J. 478
Hofri, M. 426
Holte, R.C. 480
Holzman, R. 379
Homem-de-Mello, T. 432
Hoogeveen, J.A. 73
Horowitz, E. 50, 128, 136, 331
Hu, T.C. 118, 218
Hung, M.S. 289, 292
Hurt, J. 433
Hutter, M. 171, 279

Ibaraki, T. 423
Ibarra, O.H. 97, 98, 167, 168, 234, 402
Iida, H. 412, 417, 418
Indyk, P. 431
Ingargiola, G.P. 125, 126, 202, 203
Ishii, H. 433

Jackson, R.H.F. 256
Jain, R. 318
Jaszkiewicz, A. 401
Jenkins, L. 390
Jeroslow, R.G. 91
Jia, L. 218
Johnson, D.S. 74, 96, 112, 252, 253, 298, 302, 403, 404, 411, 479, 483, 488
Johnson, E.L. 69, 121, 318, 327, 351, 455
Johnston, R.E. 217
Jokanovic, D.P. 281, 282
Jörnsten, K. 318
Jungnickel, D. 164, 221, 402, 413, 464, 465

Kalagnanam, J. 315
Kan, A.H.G. Rinnooy 442
Kann, V. 40, 381, 483
Kannan, R. 211
Karp, R.M. 97, 384
Karwan, M.H. 243, 244
Karzanov, A.V. 355
Kaskavelis, C.A. 63
Kataoka, S. 343
Katoh, N. 401, 423
Kayal, N. 327, 328, 338
Kellerer, H. 40, 46, 95, 97, 98, 101, 103, 109, 112, 113, 114, 162, 166, 167, 168, 169, 171, 173, 177, 178, 181, 207, 254, 255, 272,

273, 274, 277, 278, 279, 301, 302, 303, 304, 305, 309, 311, 395, 396, 397, 398, 399, 423, 424, 443
Kelly, J.P. 256
Keskinocak, P. 315
Khan, L.R. 217
Khan, S. 282
Khanna, S. 300, 304, 311, 313, 315, 316
Khuller, S. 316
Kim, C.E. 97, 98, 167, 168, 234, 402
Kim, I. 402
Kirkpatrick, D.G. 322
Kiwiel, K.C. 63
Klamroth, K. 393
Klein, P. 118
Kleywegt, A.J. 432, 445, 446
Knuth, D.E. 81
Koch, H. 419
Kochenberger, G.A. 262, 268, 380
Kohli, R. 232, 233
Koiran, P. 118
Kojima, M. 369
Kolesar, P.J. 127
Kolliopoulos, S.G. 408
Konno, H. 344, 346
Korbut, A. 435
Korn, R. 433
Korsh, J.F. 125, 126, 202, 203
Korte, B. 252
Kostreva, M. 390
Kothari, A. 481
Kotov, V. 443
Kovalyov, M.Y. 410
Krichen, S. 402
Krishnamurti, R. 232, 233
Kuno, T. 344, 346
Kwak, W. 390

Lagarias, J.C. 477, 478
Laguna, M. 268
Laumanns, M. 401
Lawler, E.L. 97, 98, 112, 167, 168, 182, 183, 234, 339, 410
Le, V.B. 408
Leclerc, M. 464, 465
Lee, C.F. 390
Lee, H. 390
Lee, J.S. 264, 265
Lee, K. 351
Lehmann, D. 480
Leiserson, C.E. 13, 20, 43, 60, 77, 149, 353, 377, 426, 483, 488, VII
Lenard, M.L. 118, 218
Lenstra, A.K. 477
Lenstra, H.W. 212, 477
Leung, J.Y-T. 435, 436
Lev, B. 248
Levner, E.V. 97, 98, 112, 252, 318, 338, 413
Leyton-Brown, K. 480
Li, K.F. 282
Libura, M. 434
Lin, E. 237
Lipton, R.J. 118
Liu, Y. 168
Lodi, A. 236, 450
Lorie, J.H. 237, 461
Loulou, R. 259, 264
Lovász, L. 358, 369, 477
Lueker, G.S. 306, 314, 415, 426, 427, 428, 429, 436, 440, 445

Magazine, M.J. 50, 118, 167, 218, 254, 262, 263, 265, 281
Magnanti, T.L. 186
Mahieu, Y. 352, 353, 374, 382
Malucelli, F. 362
Mamer, J.W. 429, 441
Mandel, R. 419
Manning, E.G. 282
Mansini, R. 46, 95, 97, 98, 101, 103, 109, 112, 113, 114, 465, 466, 467, 468, 469, 470, 472
Marchetti-Spaccamela, A. 40, 144, 428, 429, 430, 440, 443, 444, 445
Mardanov, S. 216
Marquardt, D.W. 63
Marsman, S. 414
Marsten, R.E. 53
Martello, S. 40, 65, 74, 88, 89, 90, 95, 96, 112, 113, 118, 121, 122, 123, 124, 127, 129, 142, 143, 144, 145, 147, 150, 197, 201, 202, 203, 204, 206, 207, 216, 217, 229, 230, 231, 232, 236, 287, 288, 289, 291, 292, 293, 296, 301, 416, 450
Martin, A. 327
Martin, C.H. 262, 264, 265, 266
Martins, E. 393
Mastrolilli, M. 171, 279
Mathews, G.B. 3, 213, 488
Mathur, K. 409
Mazzola, J.B. 240, 242
McCarl, B.A. 262
Meanti, M. 442
Megiddo, N. 238, 255, 278, 421, 422, 423

Mehrotra, A. 351
Mejia-Alvarez, P. 318
Merkle, R. 473, 474
Meyer auf der Heide, F. 118
Mezdaoui, N. 401
Michaelides, E. 259, 264
Michelon, P. 362, 382
Mingozzi, A. 292
Mohanty, B.B. 409
Monaci, M. 236, 450
Moore, G.E. 483
Morabito, R. 450
Morin, T.L. 53
Morita, H. 401, 433
Morton, D.P. 432, 433
Moser, M. 281, 282
Mosse, D. 318
Müller-Merbach, H. 121

Nagih, A. 269
Nakata, K. 369
Narasimhan, S. 247
Nash, S.G. 256
Nauss, R.M. 318, 328, 419
Neame, T. 218
Neebe, A. 292
Nehme, D. 424
Nemhauser, G.L. 53, 63, 65, 67, 68, 71, 118, 218, 221, 241, 248, 272, 351, 455, 461, 462
Ness, D.N. 237, 248, 249, 461
Neves, M.A. 452
Niemi, K.A. 403, 404
Nishida, T. 433

Odlyzko, A.M. 473, 477, 478
Oehlandt, K. 203
Ogryczak, W. 390
Oguz, O. 50, 167, 262, 263, 265, 281
Oosterhout, H. 73
Orlin, J.B. 394, 397
O'Rourke, P.J. 446
Osorio, M.A. 245, 246
Ozden, M. 249

Padberg, M.W. 70, 86, 245, 318, 327
Papadimitriou, C.H. 393, 394, 397, 480, 483
Papastavrou, J.D. 445, 446
Parajon, A. 450
Park, K. 351
Park, S. 351
Parker, R.G. 203
Parkes, D.C. 481
Parks, M.S. 433
Parra-Hernandez, R. 282, 283
Pearson, M. 480
Peled, U.N. 69, 71, 121, 245
Pérez, G. 70
Peterson, C.C. 237, 379
Pferschy, U. 40, 46, 74, 97, 98, 101, 103, 109, 112, 113, 114, 162, 166, 167, 168, 169, 171, 173, 177, 178, 181, 194, 207, 254, 255, 272, 273, 274, 277, 278, 279, 301, 302, 303, 304, 305, 309, 395, 396, 397, 398, 399, 416, 417, 418, 419, 421, 422, 423, 438, 439, 462, 465, 466, 468, 469, 470, 472
Picard, J.C. 353
Pierskalla, W.P. 240, 244
Pipher, J. 478
Pirkul, H. 238, 242, 243, 244, 245, 247, 257, 264, 265, 266, 270, 271, 282
Pirlot, M. 392, 393
Pisinger, D. 55, 60, 74, 77, 78, 79, 81, 82, 89, 93, 118, 124, 125, 127, 129, 131, 136, 137, 140, 141, 142, 143, 145, 146, 147, 154, 162, 166, 189, 191, 193, 200, 201, 202, 206, 207, 217, 227, 250, 254, 255, 272, 273, 274, 275, 277, 278, 279, 291, 292, 293, 294, 296, 298, 322, 325, 327, 330, 331, 333, 334, 339, 340, 342, 351, 356, 358, 359, 360, 362, 373, 374, 375, 378, 379, 382, 384, 414, 415, 416, 417, 418, 419, 450
Plateau, G. 88, 121, 144, 145, 243, 245, 265, 266, 269, 270, 419
Poirriez, V. 216, 222, 223, 224, 227, 229, 230
Posner, M.E. 416, 419
Potts, C.N. 190
Powell, S. 256
Pratt, V. 43
Prins, C. 73
Protasi, M. 40
Puech, C. 437, 438, 439

Rader Jr., D.J. 373, 374, 379, 380, 381, 382
Radzik, T. 422
Raidl, G.R. 268
Rajagopalan, S. 445, 446
Rajopadhye, S. 216, 222, 223, 224, 227
Ram, B. 413
Rardin, R.L. 243, 244
Rasmussen, A.B. 384

Rassenti, S.J. 479
Ratliff, H.D. 353
Ravi, R. 315
Reed, B. 434
Reilly, C.H. 271
Render, B. 186
Rendl, F. 351, 358, 367, 370, 372, 374
Resende, M.G.C. 256
Rhys, J. 351
Riha, W.O. 326, 328, 332
Rinnooy Kan, A.H.G. 259, 260, 442
Rivest, R.L. 13, 20, 43, 60, 77, 149, 353, 377, 426, 472, 478, 483, 488, VII
Rosenblatt, M.J. 390, 392
Ross, K.W. 447
Ross, S.M. 425, 447
Ryan, J. 214

Sadfi, S. 344, 347
Safer, H.M. 394, 397
Sahni, S. 50, 128, 136, 161, 165, 166, 331
Salkin, H.M. 409
Salman, F.S. 315
Salomaa, A. 473
Samphaiboon, N. 407
Sandvik, R. 384
Santos, J.L. 393
Sarin, S. 413
Savage, L.J. 237, 461
Sbihi, A. 347
Scarf, H.E. 212
Schachnai, H. 316
Schaffer, J.D. 401
Schilling, K.E. 429, 441
Schmidt, R.L. 433
Schnorr, C.P. 477
Schrader, R. 252
Schreck, H. 439
Schrijver, A. 67, 358, 369
Scott, J.P. 466
Scudder, G.D. 256, 257, 258
Seidel, R. 322
Senju, S. 257, 262
Shamir, A. 472, 475
Shapiro, A. 432
Shapiro, J.F. 221
Shaw, D.X. 402, 406, 408
Sherali, H.D. 358
Shetty, B. 350, 409, 411
Shetty, C.M. 203
Shi, Y. 390
Shih, W. 238, 247
Shiratori, N. 281, 282
Shmoys, D. 314
Shoham, Y. 480
Shoja, G.C. 282
Shub, M. 118
Silverman, J.H. 478
Simeone, B. 349, 355, 374, 379, 382, 384
Sinha, A. 318, 319, 328, 336
Sinuany-Stern, Z. 390, 392
Skiena, S.S. 5
Slivka, W. 248
Smale, S. 118
Smith, V.L. 479
Sniedovich, M. 433
Soeiro Ferreira, J. 452
Soma, N.Y. 85, 90, 91, 96
Soutif, E. 364, 367, 375
Soyster, A.L. 248
Speranza, M.G. 46, 95, 97, 98, 101, 103, 109, 112, 113, 114, 443, 466, 467
Spinrad, J.P. 408
Stair, R.M. 186
Stecke, K.E. 402
Steiglitz, K. 483
Stein, C. 13, 20, 43, 60, 77, 149, 353, 377, 426, 483, 488, VII
Steinberg, E. 433
Steiner, G. 408
Stepan, J. 433
Stewart, W.R. 256
Stougie, L. 259, 260, 442
Straus, K. 318
Sturm, J.F. 369, 384
Suhri, S. 481
Süral, H. 190
Syam, S. 350
Szkatula, K. 434, 441

Talwar, K. 480
Tamarit, J.M. 450
Tamir, A. 238, 255, 278
Tamir, T. 316
Tardos, E. 314, 480
Tarjan, R.E. 43, 355
Tecchiolli, G. 268
Teghem, J. 392, 393, 401
Teng, J.-Y. 390
Tennenholtz, M. 480
Thesen, A. 238, 247
Thiele, L. 401
Thiongane, B. 269
Thurimella, R. 316

Tien, B.N. 118, 218
Tinhofer, G. 439
Todd, M.J. 369
Toh, K.C. 369
Tonkyn, D.W. 390
Toth, P. 40, 65, 74, 88, 89, 90, 91, 95, 96, 112, 113, 118, 121, 122, 123, 124, 127, 129, 142, 143, 144, 145, 147, 150, 197, 201, 202, 203, 204, 206, 207, 216, 217, 229, 230, 231, 232, 236, 287, 288, 289, 291, 292, 293, 296, 301, 322, 351, 356, 358, 359, 360, 362, 373, 374, 375, 378, 379, 382, 384, 416
Toyoda, Y. 257, 258, 259, 262, 264, 265, 271, 282
Tremel, B. 439
Trotter, L.E. 118, 218
Tsang, D.H.K. 447
Tutuncu, R.H. 369
Tuyttens, D. 401
Tuza, Z. 443
Tzeng, G.-H. 390

Ullmann, Z. 53, 248, 461, 462
Ulungu, E.L. 392, 393, 401
Uno, T. 417, 418

Valério de Carvalho, J.M. 452
van de Geer, S. 442
van de Klundert, J.J. 408
van de Leensel, R.L.M.J. 408
van de Velde, S.L. 73, 281
van Hoesel, C.P.M. 408
van Slyke, R. 446, 447
van Wassenhove, L.N. 190
Vance, P.H. 71, 455
Vasquez, M. 268
Vaudenay, S. 478
Veilleux, L. 362, 382
Vercellis, C. 259, 260, 442, 443, 444, 445
Vickrey, W. 478
Vigo, D. 236, 450
Visée, M. 392, 393
Vizvari, B. 118
Vöcking, B. 431, 436, 437
Vohra, R.V. 479
Volgenant, A. 263, 414
Voss, S. 268

Walker, J. 283, 326, 327, 328, 332, 338
Walukiewicz, S. 121, 328, 329, 331, 424
Weatherford, L.R. 433
Weingartner, H.M. 186, 237, 248, 249, 461
Weismantel, R. 67, 71, 327, 351, 358, 367, 370, 372, 374
Weiss, H. 4
White, D.J. 233
Wiecek, M.M. 393
Willard, D.E. 60
Wirsching, G. 287
Witzgall, C. 318, 350
Woeginger, G.J. 218, 274, 302, 380, 381, 408, 416, 417, 418, 419
Wolsey, L.A. 63, 65, 67, 68, 69, 86, 120, 122, 123, 221, 241, 459
Wong, C.K. 211, 254, 339
Wood, R.K. 432, 433
Worm, J.M. 281
Wyman, F.P. 262

Yamada, T. 343, 347, 407
Yanasse, H.H. 85, 96
Yanev, N. 229, 230, 250, 251, 252
Yang, M.-H. 238
Yannakakis, M. 393, 394, 397
Young, Y. 446, 447
Yu, G. 411, 412, 424

Zak, E. 273
Zanakis, S.H. 262
Zemel, E. 43, 70, 71, 122, 140, 142, 143, 144, 145, 320, 322, 344, 346, 429
Zhu, A. 316
Zhu, N. 216
Zinober, A.S.I. 96
Zissimopoulos, V. 452
Zitzler, E. 401, 402
Zoltners, A.A. 318, 319, 328, 336
Zoon, J.A. 263
Zukerman, M. 218
Zwick, U. 43, 99

Subject Index

3-PART, 298, 302
3CNF-SATISFIABILITY, 487
O-notation, 13, 483
Θ-notation, 13

absolute performance guarantee, 31
acyclic digraph, 221
adaptive fixing, 261, 267
adaptive multimedia system, 282
adaptive threshold algorithm, 444
affine combination, 67
affinely dependent, 67
affinely independent, 67
aggregation, 212, 214, 216
AGNES, 265, 270
airline yield management, 446
algorithm
- $H^{1/(d+1)}$, 254, 255, 277
- H^{ε}, 37–40, 161, 163, 206, 254, 255, 278
- H^{ε}_{CG}, 308–310
- $H^{\frac{2}{3}}$, 302–304
- $PTAS_{CD}$, 305, 307–309
- $PTAS_{GEN}$, 310
- ABS-Comb, 472
- ABS-Ext-Greedy, 470–472
- ABS-Knap, 471, 472
- ABS-LP, 470
- B-Greedy, 187, 191, 205, 206, 212
- Backtracking-SSP, 100, 103–106, 108, 109, 111
- Backtracking, 177, 179–183
- Balanced-Search, 57–59
- Balcardssp, 276
- Balknap, 84, 139, 140
- Balmcsub, 340
- Balsub, 83, 84, 90–93, 139, 140, 340
- Basic-FPTAS, 166, 167
- Bellman-with-Lists, 75, 76, 97, 98, 100
- Bellman, 75, 80, 82, 90, 100
- Bellstate, 93, 94
- Belltab, 93, 94
- Bouknap, 192–194, 204, 205
- Branch-and-Bound, 28–30
- Branch-and-Bound(ℓ), 28, 29
- ChaillouHansenMahieu, 353
- CKPP, 161–166
- Combo, 148, 152–156, 158, 159
- Decomp, 81, 82, 89–93, 294
- Dequeue, 377
- Divide-and-Conquer, 103–108, 110, 111, 113, 114
- DP-1, 21–23, 25, 50, 52, 55
- DP-2, 23–26, 46, 49, 137, 190, 487
- DP-3, 24, 25, 76, 220
- DP-Profits, 25, 26, 41, 166, 167, 171, 191, 274, 391, 395
- DP-with-Lists, 51–55, 75, 82, 85, 90, 93, 94, 192, 249, 251, 330, 332, 436
- DP-with-Upper-Bounds, 54, 55, 156
- Dyer-Zemel, 323, 324, 331, 334, 335
- DyerKayalWalker, 328, 335, 336
- DyerRihaWalker, 333, 335–337
- EDUK, 223–227
- Emit, 133, 134
- Enqueue, 377
- Expknap, 145, 146, 152–155, 158
- Ext-B-Greedy, 205–207, 209
- Ext-Greedy, 34–39, 43, 94, 95, 161, 162, 166, 169, 254, 255, 278, 301, 391, 413, 470
- Ext-Greedy-Split, 438–440
- Find-Core, 142, 143, 146, 149, 193
- FPKP, 167, 168, 177, 178, 180–183, 195, 207
- FPSSP, 97–100, 102–104, 109, 110, 112, 114
- G-Greedy, 299
- GMS, 430, 431, 441

- Greedy, 16, 17, 29, 33, 34, 43, 45, 94, 95, 114, 118, 119, 162, 187, 205, 212, 256, 320, 428, 429, 433–435, 439, 440
- Greedy-Split, 16–18, 34, 43, 124, 162–164, 277, 435, 437, 438, 440
- Improved-Bellman, 76, 100, 103
- Improved-DP, 197, 198, 200
- Improved-Greedy, 439, 440
- Interval-Dynamic-Programming, 175–183
- KspPartition, 346
- LP-Approx, 274, 277–280
- MaxMinGreedy, 344
- Mcknap, 334, 335, 337
- MCKP-Greedy, 320–323, 326, 327, 338
- Merge-Lists, 51, 52, 55, 334
- Minknap, 146–150, 152–156, 158, 159, 192, 250, 294, 295, 335, 384, 419, 472
- MT1, 129, 144, 145, 203, 230, 294
- MT2, 144, 145, 147, 150, 152–155, 158, 204, 231
- MTB, 203–205, 230
- MTB2, 204
- MTBhard, 204, 205
- MThard, 145, 152–155, 158, 159, 204
- MTM, 292–294, 296–298
- MTS, 88, 89
- MTSL, 89–93
- MTU1, 230, 231
- MTU2, 230, 231
- Mulknap, 294–298
- Primal-Branch, 128–130
- Primal-Dual-Branch, 129, 130, 146
- QuadBranch, 376, 377
- QuadKnap, 351, 375, 382, 383
- Quicksort, 142
- Relaxed-Dynamic-Programming, 100–106, 109, 111
- Recursion, 103–106, 177–183
- Recursive-DP, 46–50, 76, 178
- Scaling-Reduction, 169–171, 175, 177, 181–183, 279, 280
- Split, 43, 44, 289, 320, 322, 346
- ST00, 90–93
- Threshold, 444–446
- TV-Greedy, 232, 233
- U-Greedy, 212, 232, 233
- Unbounded-DP, 219–222
- Unsorted-Greedy, 434, 435
- VectorMerge-Interval, 173–176
- Wordmerge, 133–136, 331, 340
- Wordsubsum, 78, 90, 93, 94
- WordsubsumPD, 81, 91–93

all-capacities knapsack problem, 9, 20, 21, 61, 130, 131, 137, 162, 191, 194, 220, 223, 275
all-pairs shortest path problem, 9
almost strongly correlated instance, 143, 151
amortization, 467–469
approximability, no *FPTAS*
- multidimensional knapsack problem, 252–254
- multiple knapsack problem, 301–302
- quadratic knapsack problem, 380

approximate dynamic programming, 250, 266–268
approximation algorithm, 29–36
- $(1-\varepsilon)$–approximation algorithm, 33
- k–approximation algorithm, 33
- asset-backed securitization, 469–472
- cardinality constrained knapsack problem, 276–280
- class-constrained multiple knapsack problem, 316
- knapsack problem, 15–17, 33–36, 161–183
- multidimensional knapsack problem, 252–255
- multiobjective knapsack problem, 393–400
- multiple knapsack problem, 298–315
- multiple knapsack problem with assignment restrictions, 315
- multiple-choice knapsack problem, 338–339
- quadratic knapsack problem, 380–381
- subset sum problem, 94–96

approximation of the Pareto frontier, 393
approximation ratio, 33
approximation scheme, 37–42
ε–approximation scheme, 37–40
arc of a graph, 221
arithmetic rounding, 171, 279
arrival time of an item, 445
asset-backed securitization, 466–472
asymptotic running time bounds, 13
asymptotic upper bound, 12
average running time, 12
avis instance, 91–94, 112–115

backward greedy solution, 125
balanced complete bipartite subgraph problem, 380
balanced filling, 55

balanced operations, 55
balanced solution, 54
balancing, 54–60
– bounded knapsack problem, 200
– fixed-cardinality subset sum problem, 275
– knapsack problem, 138–140
– multiple-choice knapsack problem, 331
– multiple-choice subset sum problem, 339–340
– subset sum problem, 56–60, 82–85
bang-for-buck ratio, 257
Bellman recursion, 21, 60, 131
Bernoulli distribution, 431
best-first search, 29
bidimensional knapsack problem, 269
bin packing, 74, 236, 443, 451, 455
binary decision, 1
binary search, 484
binary tree, 165, 180, 182
binary variable, 1
BIP-DECISION, 488, 490, 491
bipartite graph, 402
bit strings, 22
bottleneck multiple subset sum problem, 286
bound-and-bound, 292
bounded knapsack problem, 6, 185–209
– capital budgeting, 462
– two-stage cutting problem, 451–452, 454
bounded knapsack problem with setup costs, 190
bounded subset sum problem, 200
branch-and-bound, 27–29, 53–54
– bounded knapsack problem, 201–204
– cardinality constrained knapsack problem, 273
– knapsack problem, 119, 127–130
– multidimensional knapsack problem, 246–247, 267
– multiobjective knapsack problem, 393
– multiple knapsack problem, 292–298
– multiple-choice knapsack problem, 328–329
– precedence constraint knapsack problem, 408
– quadratic knapsack problem, 374–375
– subset sum problem, 85–87
– unbounded knapsack problem, 214, 228–232
branch-and-bound tree, 28, 30, 53, 247, 267, 428
branch-and-cut, 272
break item, *see* split item
budget planning, *see* capital budgeting
bundle methods, 63

capacity, 2
capacity dense (MSSP), 304
capacity grouped (MSSP), 304, 308
capital budgeting, 186, 248, 281, 318, 343, 390, 409, 411, 461, 462
capital budgeting problem, 237
CARDINALITY (2-KP), 253
cardinality bound, 229, *see* cardinality constraint
cardinality constrained knapsack problem, 122, 237, 269, 271–280, 316
cardinality constraint, 122, 193, 201, 202, 237, 271, 414, 431
cargo loading, 287
certificate, 485
change-making problem, 415–416
Chvatal-Gomory cut, 358
circle instance, 157–159
class-constrained knapsack problem, 316
class-constrained multiple knapsack problem, 315–316
clearing house, 464
clearing run, 464
clearing system, *see* interbank clearing system
CLIQUE, 350, 403, 424, 487
closed directed path, 403
collapsing knapsack problem, 274, 416–419
collective dominance, 216, 225, 227
color of an item, 315
column generation, 342, 455–459
combinatorial auction, 478–482
combinatorial auction problem, 478
compartment of a knapsack, 316
competitive analysis, 443
competitive ratio, 443
complementary slackness, 246
complete graph, 402
componentwise addition, 51
composite dual problem, 241–244
composite heuristic, 233
composite relaxation, 240–244
computational results
– asset-backed securitization, 472
– bounded knapsack problem, 204–205
– knapsack problem, 150–160
– multidimensional knapsack problem, 243
– multiple knapsack problem, 296–298

- multiple-choice knapsack problem, 335–337
- quadratic knapsack problem, 382–388
- subset sum problem, 90–94, 112–114
- two-dimensional knapsack problem, 270–271
computer
- AMD Athlon, 1.2 GHz, VIII, 91, 92, 94
- Intel Pentium 4, 1.5 GHz, VIII, 204, 205, 336, 337, 383
- Intel Pentium III, 933 MHz, VIII, 152, 153, 155, 158, 159, 296, 297, 384–387
- Intel Pentium, 200 Mhz, 113
$(1,k)$-configuration, 70
$(1,k)$-configuration inequality, 70
constraint pairing, 245–246, 374
container loading, 450
continuous quadratic knapsack problem, 352
convex combination, 67
convex hull, 68, 242, 322, 408
convex set, 67
core
- bounded knapsack problem, 191–203
- equality knapsack problem, 414
- knapsack problem, 140–147, 429, 430
- multiple-choice knapsack problem, 332
- subset sum problem, 88–90
- unbounded knapsack problem, 194, 230–232
core algorithm, *see* core
core problem, 89, 141
cover, 69, 459
cover inequality, 459
CPLEX, 271, 384, 472
credit, 462–464
credit rationing, 462
creditworthiness, 462
critical item, *see* split item
cutting pattern, 455
cutting problem, 186
cutting stock problem, 455
cutting technology, 449

Dantzig bound, 120
Dantzig-Wolfe decomposition, 342
d-dimensional knapsack problem, *see* multidimensional knapsack problem
decision problem, 484
- 3-PART, 298, 302
- 3CNF-SATISFIABILITY, 487
- BIP-DECISION, 488, 490, 491
- CARDINALITY (2-KP), 253
- CLIQUE, 350, 403, 424, 487
- EQUIPARTITION, 252, 253, 298, 299
- FORMULA-SATISFIABILITY, 487, 488
- HAM-CYCLE, 488
- KP-DECISION, 484, 485, 487, 488, 490, 491, 493
- KP-OPTIMIZATION, 484, 486
- MCKP-DECISION, 493
- QKP-DECISION, 487, 493
- SET-COVERING-DECISION, 488
- SSP-DECISION, 74, 75, 80, 85, 86, 88, 92, 93, 156, 381, 415, 473, 487, 488, 490–492
- TSP-DECISION, 488
- UKP-DECISION, 493
- USSP-DECISION, 491–493
decreasing, VII
Dembo and Hammer bound, 126, 138, 145, 193, 291, 327
denominations of coins, 415
dense subgraph problem, 350
density of a subset sum problem, 477
density of an instance, 382
dependent set, 69
depth-first search, 29, 127, 271, 403
Deutsche Bundesbank, 465
diagonal matrix, 368
diagonal profit matrix, 350
digraph, 381
dimension of a polyhedron, 68
1.5–dimensional knapsack problem, 269, 272
directed circuit, 403
directed path, 403
dissimilar capacities instance, 296
distributed computer system, 238
diversification, 466
divide and conquer, 46, 179, 180
dominance
- knapsack problem, 50–53, 436, 437
- multidimensional knapsack problem, 244, 249, 250, 283
- multiobjective knapsack problem, 389
- multiple-choice knapsack problem, 319
- stochastic knapsack problem, 433
- subset sum problem, 58
- unbounded knapsack problem, 214, 216–219, 223–227
dual greedy heuristic, 256–260, 281, 465
dual problem, 245, 260
Dutch highway authority, 281

dynamic and stochastic knapsack problem, 445, 447
dynamic programming, 20, 50–54
– bounded knapsack problem, 190–200, 207–209
– cardinality constrained knapsack problem, 273–276, 279–280
– knapsack problem, 20–26, 130–140, 147–150, 166–183
– knapsack sharing problem, 343
– multidimensional knapsack problem, 248–252, 266–268
– multiobjective knapsack problem, 391–393
– multiple-choice knapsack problem, 329–331
– precedence constraint knapsack problem, 404–408
– subset sum problem, 75–76, 79–80, 85
– two-stage cutting problem, 452
– unbounded knapsack problem, 219–228, 233–234
dynamic programming by profits, 25, 166, 175–183, 191, 207, 208, 222, 233, 274, 329, 338, 391
dynamic programming by reaching, 25
dynamic programming by weights, 21, 274
dynamic programming with lists, 50–53

edge series parallel graph, 380
efficiency, 15, 257, 427, 469
efficient solution, 389
electronic commerce, 478
equality knapsack problem, 413–414
EQUIPARTITION, 252, 253, 298, 299
evenodd instance, 91–94
evolutionary algorithm, 237, 268
exact k-item knapsack problem, 272
exact k-item subset sum problem, 273, 275
exact subset sum problem, 472–478
expanding core
– bounded knapsack problem, 191–194
– generalized multiple-choice knapsack problem, 342
– knapsack problem, 145–147
– multiple-choice knapsack problem, 330, 332–335
– subset sum problem, 89
expanding knapsack problem, 416
expected performance, 31
exponential distribution, 431
extended cover inequality, 69
extension for a minimal cover, 69

face, 68
face defining inequality, 68
facet, 68, 272
feasible knapsack filling, 306
fill-up and exchange operations, 379
financial decision problems, 461–465
first-fit-decreasing, 301
fixed core, 89, 144–145
– multiple-choice knapsack problem, 334
fixed size core, 90
fixed variables, 119
fixed-cardinality subset sum problem, 275
fixed-core algorithm, 147, 203
formula satisfiability problem, 488
FORMULA-SATISFIABILITY, 487, 488
forward greedy solution, 125
Fourier-Motzkin elimination, 246
FPTAS, *see* fully polynomial time approximation scheme
fractional knapsack problem, 421–423
free variables, 28, 119, 126, 273
freight dispatcher, 445
French amortization, 467
full-dimensional polyhedron, 68
fully polynomial time approximation scheme, 40
– bounded knapsack problem, 207–209
– cardinality constrained knapsack problem, 279–280
– class-constrained knapsack problem, 316
– knapsack problem, 42, 166–183
– multiobjective knapsack problem, 395–397
– multiobjective optimization problem, 394
– multiple-choice knapsack problem, 338
– Pareto frontier, 393
– subset sum problem, 97–112
– unbounded knapsack problem, 233–234

gap problem of a multiobjective optimization problem, 394
general integer programming, 483
generalized assignment problem, 286, 318
generalized greedy algorithm, 299
generalized multiple-choice knapsack problem, 340–342
genetic algorithm, 268, 401
geometric constraint, 236

geometric rounding, 171, 279
glass cutting, 449
Gomory cut, 248
greedy choice property, 17
greedy-type algorithm, 444
– asset-backed securitization, 469–471
– bounded knapsack problem, 203, 206–209
– expected performance, 433–436
– interbank clearing system, 465
– knapsack problem, 161–165, 177
– knapsack sharing problem, 344
– multidimensional knapsack problem, 256–260
– multiobjective knapsack problem, 392
– multiple knapsack problem, 299–301
– multiple knapsack problem with assignment restrictions, 315
– multiple-choice knapsack problem, 319–321, 328
– quadratic knapsack problem, 379
– subset sum problem, 94–95, 437
– unbounded knapsack problem, 222, 233
guillotine cut, 449

HAM-CYCLE, 488
Hamilton cycle problem, 488
harmonic series, 260
heuristic
– dual, 379
– multidimensional knapsack problem, 255–268
– multiple-choice knapsack problem, 338–339
– primal, 379
– quadratic knapsack problem, 379–380
hill-climbing algorithms, 480
Horowitz and Sahni decomposition, 50, 81–82, 136, 293
hybrid algorithm
– multidimensional knapsack problem, 264
– multiple-choice knapsack problem, 332
– subset sum problem, 87–88

ill-conditioned, 151
in-tree, 403
increasing, VII
inner product between matrices, 368
input size, 484
instance, 2
– almost strongly correlated, 143, 151
– avis, 91–94, 112–115
– circle, 157–159
– dissimilar capacities, 296
– evenodd, 91–94
– formal definition, 484
– inverse strongly correlated, 143, 151
– monotone, 336
– mstr, 157–159
– p14, 112–114
– pceil, 157–159
– psix, 91–94
– pthree, 91–94
– similar capacities, 296
– somatoth, 91–94
– span, 156–159
– strongly correlated, 143, 151, 204, 296, 336
– subset sum, 143, 152, 204, 296, 336
– todd, 112–115
– uncorrelated, 143, 150, 204, 296, 335
– uncorrelated similar weights, 152
– weakly correlated, 143, 150, 204, 296, 336
– zig-zag, 336
integer diagonal quadratic knapsack problem, 350
integer knapsack problem, 6
integrality gap, 427–429
interbank clearing system, 464–465
interbank rate, 463
interest rate, 466, 467
inverse strongly correlated instance, 143, 151
inverse-parametric knapsack problem, 421
investment problem, 186, 433, 461
Italian amortization, 468
item, 2
item set, 2
item type, 185, 211

k-item knapsack problem, *see* cardinality constrained knapsack problem
k-item subset sum problem, 273
knapsack cryptosystem, 472–478
knapsack polytope, 68–72, 459
knapsack problem, 2
– $\mathcal{NP}$-hardness, 491
– approximation algorithm, 161–183
– asset-backed securitization, 469, 471
– column generation, 457
– exact solution, 117–160

- fully polynomial time approximation scheme, 166–183
- network formulation, 392
- polynomial time approximation scheme, 161–166
knapsack problem with generalized upper bound constraints, 317
knapsack sharing problem, 342–347
KP-DECISION, 484, 485, 487, 488, 490, 491, 493
KP-OPTIMIZATION, 484, 486

Lagrangian decomposition, 65–67, 270, 362–367
Lagrangian dual problem, 63, 442
- knapsack problem, 123
- multidimensional knapsack problem, 241–244, 442
- multiple-choice knapsack problem, 326
- quadratic knapsack problem, 354, 360, 362, 367
- two-dimensional knapsack problem, 269
Lagrangian multiplier, 62, 213, 239–244, 262–264, 269, 281, 282, 361, 442
Lagrangian profit, 361, 363
Lagrangian relaxation, 62–65
- cardinality constrained knapsack problem, 272
- multidimensional knapsack problem, 238–245, 262–264, 442
- multidimensional multiple-choice knapsack problem, 282
- multiple knapsack problem, 63–64, 289–290, 292
- multiple-choice knapsack problem, 325–328
- quadratic knapsack problem, 352–355, 361–363, 373, 374
- two-dimensional knapsack problem, 269–270
leaf of a tree, 165
lease asset, 466
leasing, 466
least common multiple, 215
left-right approach, 404
level of a node, 48, 111
level-oriented packing, 450
LEX heuristic, 379
lifting, 70
linear combination, 67
linear decision model, 1
linear generalized multiple choice knapsack problem, 341
linear majorization function, 355
linear median algorithm, 182, 231
linear multiple-choice knapsack problem, 317
linear programming relaxation, *see* LP-relaxation
linear quadratic knapsack problem, 352
linearly dependent, 67
linearly independent, 67
load of a knapsack, 286
loading problem, 186
loan, 462, 466
London transport, 478
longest path problem, 221
longest processing time, 302
Lovász lattice reduction algorithm, 477–478
lower bound
- knapsack problem, 124–125
- multidimensional knapsack problem, 245, 250–252
- multiple knapsack problem, 293
LP-dominance, 319
LP-extreme, 320
LP-relaxation, 17, 62
- asset-backed securitization, 469–470
- bounded knapsack problem, 186–187, 201, 203
- cardinality constrained knapsack problem, 272, 273, 276–280
- cutting stock problem, 455, 456
- fractional knapsack problem, 423
- generalized multiple-choice knapsack problem, 342
- knapsack problem, 17–19, 25, 28, 120, 427, 428, 435
- knapsack sharing problem, 344–345
- multidimensional knapsack problem, 238, 241–243, 245–246, 248, 251, 254–256, 260–262, 265
- multiobjective knapsack problem, 399
- multiperiod knapsack problem, 424
- multiple knapsack problem, 65, 289, 299
- multiple-choice knapsack problem, 319–325
- quadratic knapsack problem, 352, 356–359, 361, 362
- two-dimensional knapsack problem, 269
- unbounded knapsack problem, 212

maintenance, 281

marble production, 287
Markov decision process, 446
Markowitz model, 463
master problem, 456
Matlab, 384
max clique, 350
max-min knapsack problem, 411–412
maximum flow problem, 353
MCKP-DECISION, 493
median, 182, 231
median algorithm, 28, 43
menu planning, 318
merging of lists, 51–53
metaheuristic, 237, 268, 401–402
minimal core, 194
minimal cover, 69, 459
minimal cover inequality, 69
minimization knapsack problem, 412–413, 460
minimization unbounded knapsack problem, 218
money transfer, 464–465
monotone instance, 336
monotonicity properties, 345
monotonicity property, 162, 172, 227
Monte Carlo simulation, 432
mortgage, 466
MOSA method, 401
mstr instance, 157–159
multi-period decision problem, 281
multi-unit combinatorial auction, 479–482
multiconstraint knapsack, 236
multidimensional knapsack problem, 6, 235–283, 491
– asset-backed-securitization, 467
– combinatorial auction, 480
– interbank clearing system, 465
– probabilistic model, 440–442, 447
multidimensional multiple-choice knapsack problem, 249, 280–283, 469
multiobjective d-dimensional knapsack problem, 390
multiobjective knapsack problem, 389–402
multiobjective linear discrete optimization problems, 394
multiobjective tabu search method, 401
multiperiod knapsack problem, 424
multiple dominance, 216, 225, 232
multiple knapsack problem, 7, 236, 285–316
multiple knapsack problem with assignment restrictions, 315
multiple knapsack problem with identical capacities, 7, 285
multiple loading problem, 287
multiple subset sum problem, 8, 285
multiple subset sum problem with identical capacities, 286
multiple-choice knapsack problem, 8, 237, 280, 317–347
– $\mathcal{NP}$-hardness, 493
– capital budgeting, 462
– combinatorial auction, 482
multiple-choice subset sum problem, 339
multiplier search branch-and-bound algorithm, 411
multiply-iterated cryptosystem, 475

native constants, 62, 136
network flow problem, 353
node of a tree, 165
non-approximability, *see* approximability
non-polynomial algorithms, 13
nondecreasing, VII
nonincreasing, VII
nonlinear knapsack problem, 318, 409–411
nonlinear resource allocation problem, 409
normal distribution, 432, 433
$\mathcal{NP}$-complete, 486
$\mathcal{NP}$-hard, 486, 487
$\mathcal{NP}$ problems, 486
NTRU cryptosystem, 478

on-line algorithm, 443
on-line knapsack problem, 426, 442–447
open node, 57
optimal policy, 446
optimal solution set, 3, 46
optimal solution value, 3
optimal solution vector, 3
optimal substructure, 20, 51, 53
oriented cut, 449
out-tree, 403
outstanding principal, 467
over-filled knapsack, 120
overbooking, 446
overflow probability, 431

p14 instance, 112–114
pairing of constraints, 245–246
paper roll, 452
parallel composition, 381
parametric knapsack problem, 419–421

parametric profit knapsack problem, 420
parametric weight knapsack problem, 420
Pareto curve, 390
Pareto frontier, 390
Pareto optimal solution, 389
Pareto simulated annealing, 401
partial enumeration, 121, 431
partially ordered knapsack problem, 402
pceil instance, 157–159
penalty term, 63
penalty value, 259, 432
performance of algorithms, 11–14
periodicity, 214–216
pivot operation, 262
Poisson distribution, 427, 432
Poisson process, 427, 446, 447
polyhedron, 68
polynomial time algorithms, 13
polynomial time approximation scheme, 40
– bounded knapsack problem, 206–207
– cardinality constrained knapsack problem, 278–279
– class-constrained multiple knapsack problem, 316
– knapsack problem, 40, 161–166
– multidimensional knapsack problem, 254–255
– multiobjective knapsack problem, 397–400
– multiple knapsack problem, 311–315
– multiple subset sum problem, 304–310
– Pareto frontier, 393
– stochastic knapsack problem, 431
– subset sum problem, 94–96
– unbounded knapsack problem, 233
polynomially solvable problems, 485
polytope, 68
pool of open nodes, 57, 82
portfolio, 468
portfolio optimization, 433
portfolio selection, 464
positive semidefinite, 368
$\mathcal{P}$ problems, 485
precedence constraint knapsack problem, 402–408
primal greedy heuristic, 256–260, 264–265, 465, 480
primal-dual algorithm, 79, 128
– branch-and-bound, 129–130, 145
– dynamic programming, 145
– knapsack problem, 136–140
– multiple-choice knapsack problem, 330
– subset sum problem, 79–80
– word RAM algorithm, 80–81
private key, 473
probabilistic model, 12
problem
– (kKP), 122, 124, 271–280, 316, 418, 419
– (kSSP), 273, 275
– (2-KP), 236, 239, 243, 252–254, 269, 270, 397, 463
– (B-MSSP), 286, 298, 299, 301–304
– (BKP), 6, 168, 185–194, 197–207, 209, 211, 212, 214, 219, 220, 222, 228–230, 232–234, 317, 451, 452, 454, 462
– (CCKP), 316
– (CCMKP), 315, 316
– (CKP), 416–419
– (CMP), 415, 416
– (E-kKP), 272–275, 277, 278, 280
– (E-kSSP), 273, 275, 276
– (EKP), 413
– (ESSP), 473, 474
– (FKP), 421–423
– (GAP), 286, 287, 314
– (GMCKP), 340–343
– (KP), 2–6, 9, 10, 15–17, 19–23, 25, 28, 30–33, 44, 45, 49–51, 53–56, 64, 65, 73–75, 81, 88, 94, 95, 117–122, 124–127, 130, 131, 133, 137, 138, 140, 144, 149, 150, 152, 154, 161, 163, 164, 166–169, 171, 173, 175, 176, 181, 183, 186–191, 194, 195, 200–208, 212, 214, 218, 220, 222, 228, 230–232, 234–240, 243–245, 247, 248, 250, 254–256, 264, 265, 269, 270, 272, 274, 277–280, 283, 285, 288, 293, 295, 298–300, 316, 318, 320, 327, 330, 331, 334, 336, 338, 339, 343, 345, 350, 355, 361, 395–397, 402, 408–414, 416–419, 422–424, 428, 429, 431, 436, 440, 443, 462, 467, 469–472
– (KPR), 126
– (KSP), 343, 346, 347
– (LKP), 17–19, 25, 28, 29, 43, 54, 120–124, 359, 375, 376
– (MCKP), 8, 189, 190, 237, 280, 281, 283, 317–322, 327–331, 333, 334, 336, 338–340, 342–344, 462, 482
– (MCSSP), 339
– (MKP), 7, 8, 63, 65, 285–289, 291, 292, 298–301, 304, 311, 313, 315, 316
– (MKP-I), 7, 285, 300, 304
– (MKPAR), 315
– (MMKP), 411, 412
– (MOKP), 390–393, 395, 397, 401, 402
– (MOd-KP), 390, 397, 399

- (MPKP), 424
- (MSSP), 8, 285, 287, 300, 304–310, 313
- (MSSP-I), 8, 286, 298, 300, 301, 304
- (MaxMinMCKP), 343
- (MinKP), 412, 413
- (NLK), 409–411
- (PCKP), 402–404, 406–408
- (PPKP), 420, 422
- (QKP), 8, 9, 349–352, 355, 356, 360, 362, 364, 367, 369–371, 373, 374, 376, 379–381, 384
- (SCKP), 414, 415
- (SSP), 5, 56, 57, 60, 73–76, 79–81, 83–87, 89–91, 94–100, 109, 112, 118, 119, 182, 200, 273, 291, 295, 305, 336, 340, 414, 426, 437–439, 473
- (SUKP), 423, 424
- (UKP), 6, 51, 211–214, 216–223, 226, 229–234, 244, 249, 454, 462
- (VM), 172–176
- (d-KP), 7, 235–240, 242, 244–252, 254–256, 260–262, 264–271, 278, 280–283, 390, 397, 399, 424, 426, 440–442, 447, 462, 465, 467, 480
- (d-MCKP), 280, 281, 283, 469
- GMCKP, 341
- KP, 120, 131, 141
- MSSPAR, 315
- QP, 363
- SSP, 86, 87
- UCCMKP, 316

profit, 2
profit density, 442
profit matrix, 349
profit to weight ratio, 15
proper face, 68
prune and search method, 322
pseudopolynomial algorithms, 13
pseudopolynomial time, 487
psix instance, 91–94
PTAS, *see* polynomial time approximation scheme
pthree instance, 91–94
public housing, 466
public key, 473

QKP-DECISION, 487, 493
quadratic 0-1 programming problem, 353
quadratic assignment problem, 362
quadratic knapsack problem, 8, 349–388
- $\mathcal{NP}$-hardness, 487, 493
- generalization to graphs, 380

quality of service, 282
quasi-subgradient, 243

RAMBO, 245
random access machine, 118
rank of a matrix, 368
rating, 463
reachable intervals, 100
reachable values, 75, 100
reactive tabu search, 268
rebalancing, 165
recursion depth, 183
recursive structure, 29, 168, 180, 182, 183
reduced reachable values, 100
reduced solution set, 390
reduction, 44–45
- knapsack problem, 125–127
- multidimensional knapsack problem, 244, 247, 249
- multiple knapsack problem, 291–292
- multiple-choice knapsack problem, 327–328, 331–332
- quadratic knapsack problem, 373–374

reduction to a decision problem, 486
relative performance guarantee, 33
relaxation, 62–65, 238–246
relaxed dynamic programming, 100
relevance value, 257
relevant items, 98, 99, 305
restricted master problem, 457
restricted RAM model, 62
return on equity, 466
risk, 433
risk grade, 462
risk-free investment, 463
road maintenance, 281
root of a tree, 165, 403
running time, 11
running time bounds, 12

SADE, 270
scaling, 41, 166–171, 175, 181, 182, 207–209, 233, 338
SeDuMi semidefinite solver, 384
semi-on-line problem, 443
semidefinite optimization problem, 368
semidefinite relaxation, 369
separability property, 50, 331
separation of cover inequalities, 459–461
separation problem, 459

sequential lifting procedure, 70, 408
series composition, 380, 381
set covering problem, 488
SET-COVERING-DECISION, 488
set-union knapsack problem, 423–424
setup costs, 190
setup operation, 189
shifting technique, 306, 313
similar capacities instance, 296
simple dominance, 216, 249
simulated annealing, 237, 268, 401
simulation, 31
simultaneous lifting procedure, 70
single-source shortest path problem, 9
single-unit combinatorial auction, 479
singly-iterated cryptosystem, 475
skew-symmetric matrix, 360
slope, 322
somatoth instance, 91–94
source of a graph, 381
span instance, 156–159
sparse computation, 227
special purpose vehicle, 466
split class, 320
split item, 17, 43–44, 79, 186, 201, 299, 320, 408, 429, 430, 437, 470
split item type, 186, 201
split solution, 18, 55–57, 79, 120, 321, 330, 338, 340
SSP-DECISION, 74, 75, 80, 85, 86, 88, 92, 93, 156, 381, 415, 473, 487, 488, 490–492
state, 50–54, 131, 138, 332
state reduction, *see* reduction
steel coil, 452
steel cutting, 449
stochastic knapsack problem, 426
stochastic process, 447
storage management for multimedia systems, 316
storage reduction scheme, 46–50, 178, 180, 191, 200, 219, 249, 275
strength Pareto evolutionary algorithm, 401
strip packing, 451
strong duality, 245
strong minimal cover, 70
strongly $\mathcal{NP}$-hard, 487
strongly correlated instance, 143, 151, 204, 296, 336
strongly correlated knapsack problem, 122, 125, 141, 414–415
subgradient optimization, 243–244, 269, 360, 364, 367
subset sum instance, 143, 152, 204, 296, 336
subset sum problem, 5, 56–62, 73–114
– $\mathcal{NP}$-hardness, 487–491
– probabilistic model, 437–440
super-modular knapsack problem, 350
superincreasing sequence, 474
supported efficient solutions, 391
supporting face, 68
surrogate dual problem, 65
– knapsack problem, 124
– multidimensional knapsack problem, 241–244, 249
– two-dimensional knapsack problem, 269, 270
surrogate multiplier, 239–244, 247, 257, 264–265, 270
surrogate relaxation, 62, 64–65
– knapsack problem, 124
– max-min knapsack problem, 412
– multidimensional knapsack problem, 239–245, 247, 264–265
– multiple knapsack problem, 64–65, 288, 293
– strongly correlated knapsack problem, 414
– two-dimensional knapsack problem, 269
surrogate subgradient methods, 63

tabu search, 268, 380, 401
target sum, 415
telecommunication systems, 447
terminal of a graph, 381
three-dimensional packing, 450
threshold accepting algorithm, 237
threshold dominance, 216, 227
threshold efficiency, 444
threshold policy, 446
tightness, 33
time dependent knapsack problem, 445
todd instance, 112–115
total capacity of knapsacks, 286
total profit, 15
total weight, 15
total-value greedy heuristic, 232
transformation to a decision problem, 486
transmission capacity, 447
transportation request, 445, 446
traveling salesman problem, 483, 488
tree of balanced solutions, 57
trimming, 450
trust region methods, 63
truthful combinatorial auction, 481

TSP-DECISION, 488
two-dimensional bin packing problem, 450
two-dimensional cutting problem, 449–454
two-dimensional knapsack problem, 269–271, 463, 464
two-dimensional packing problem, 236, 450
two-dimensional strip packing problem, 451
two-stage cutting pattern, 449
two-stage cutting problem, 449–454

UKP-DECISION, 493
unbounded knapsack problem, 6, 211–234
– $\mathcal{NP}$-hardness, 493
– capital budgeting, 462
– two-stage cutting problem, 452, 454
unbounded subset sum problem, 227
uncorrelated instance, 143, 150, 204, 296, 335
uncorrelated similar weights instance, 152
uniform distribution, 426, 432, 434–436, 438–440, 442
update operation, 171, 175, 194–199, 220
upper bound, 27, 53–54, 62–64
– cardinality constrained knapsack problem, 273, 274
– knapsack problem, 119–124, 147–150
– multidimensional knapsack problem, 245, 247, 250–252, 263–264
– multiple knapsack problem, 288–293
– multiple-choice knapsack problem, 319–327
– quadratic knapsack problem, 351–373, 376–378, 384–388
– subset sum problem, 86–87
upper plane, 355, 359
USSP-DECISION, 491–493

valid inequality, 68, 408
value-pseudo-polynomial, 394
variable reduction, *see* reduction
vector evaluated genetic algorithm, 401
vector merging problem, 168, 171, 175
vector of diagonal elements, 368
vector product, 368
verification algorithm, 485
vertex of a graph, 221
vertex series parallel graph, 380
VLSI design, 318

weak dominance, 389
weakly $\mathcal{NP}$-hard, 487
weakly correlated instance, 143, 150, 204, 296, 336
weight, 2
weight inequalities, 71
weighted graph, 221
weighted sum scalarization, 391, 393, 401
winner determination, 478
wood cutting, 449
word RAM algorithm, 60–62
– bounded knapsack problem, 200
– knapsack problem, 131–136
– multiple choice subset sum problem, 340
– multiple-choice knapsack problem, 331
– primal-dual, 80–81
– subset sum problem, 60–62, 76–79
– unbounded subset sum problem, 227
word RAM model of computation, 60
word size, 60, 227
word-parallelism, 60
worst-case analysis, 12
worst-case behaviour, 31

yield management, 446

zig-zag instance, 336