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>.

springeronline.com

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

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

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.

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.

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

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:

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

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

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

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 $p_{\max} = \max\{p_j \mid j = 1,\ldots,n\}$ and $w_{\max} = \max\{w_j \mid j = 1,\ldots,n\}$ denote the largest profit and the largest weight value, respectively. Analogously, $p_{\min} := \min\{p_j \mid j = 1,\ldots,n\}$ and $w_{\min} := \min\{w_j \mid j = 1,\ldots,n\}$ 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

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

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

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

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

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 $\{1,\ldots,j\}$ 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 $q = 1,\ldots,U$, all other values can be computed for $j = 1,\ldots,n$ and $q = 0,\ldots,U$ 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 $U = U_{\text{LP}} = 16$. 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

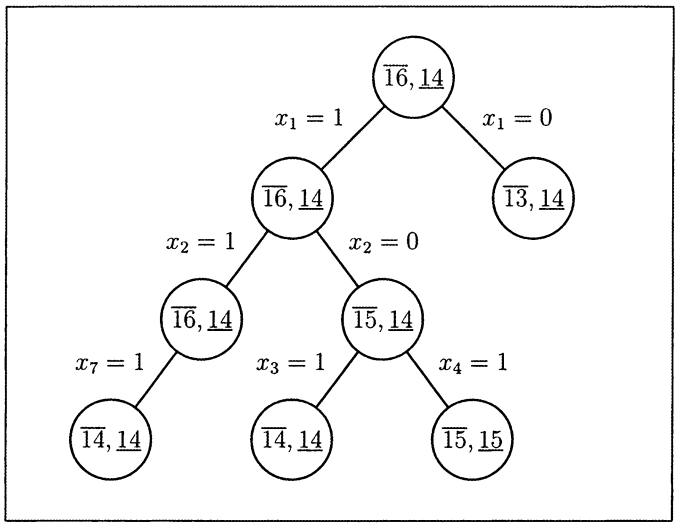

Fig. 2.9. A tree illustrating Branch-and-Bound applied to our example.

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 :

$\ell := \min\{\lceil \frac{1}{\varepsilon} \rceil - 2, n\}$
$z^A := 0$ 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
      $z^A := \sum_{j \in L} p_j$ *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.

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.

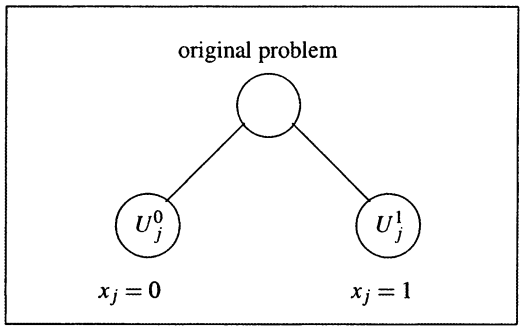

Fig. 3.2. Branching on the two possible solution values of x_j.

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

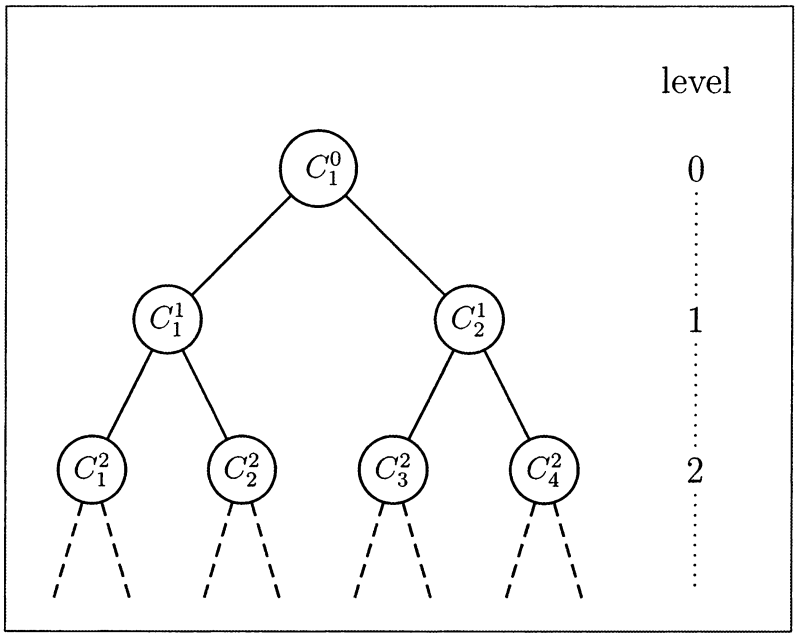

Fig. 3.4. A binary rooted tree representing the recursive structure of Recursive-DP by nodes labeled with the capacity parameters.

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

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

since $g(n,\overline{w}) \le n$ 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 $W = 8$.

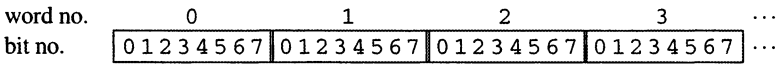

Fig. 3.10. Numbering of words and bits

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

	0 1 2 3 4 5 6 7 8 9
$g(1,\cdot)$	1 0 0 1 0 0 0 0 0 0
$g'(1,\cdot)$	0 0 0 0 0 1 0 0 1 0
$g(2,\cdot)$	1 0 0 1 0 1 0 0 1 0

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 $(\mu_1,\ldots,\mu_m)$.

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

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*

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

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)

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

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

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

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 $N = \{1,\ldots,n\}$ of n items with positive integer weights $w_1,\ldots,w_n$ 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:

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

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

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

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.,

$$w_j \leq c, \quad j = 1,\ldots,n, \tag{4.4}$$

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.,

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

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]).

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

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

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:

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

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

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

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

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

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

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

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 $L(\mathrm{SSP},\lambda)$

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

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
	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
Belltab	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	▫	▫
	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
Bellstate	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	▫	▫
	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
Wordsubsum	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 $\Delta_{\tilde{\delta}}$. 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*

$$z^A \geq (1-\varepsilon)c \quad \textit{or} \quad z^A = z^*.$$

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 $y^{\tilde{B}}$ 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

$$\sum_{i=1}^{m_\ell} \frac{\rho}{2^\ell} \cdot \frac{\tilde{c}_\ell^i}{\varepsilon c} \leq \frac{\rho}{2^\ell} \cdot \frac{1}{\varepsilon}.$$

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