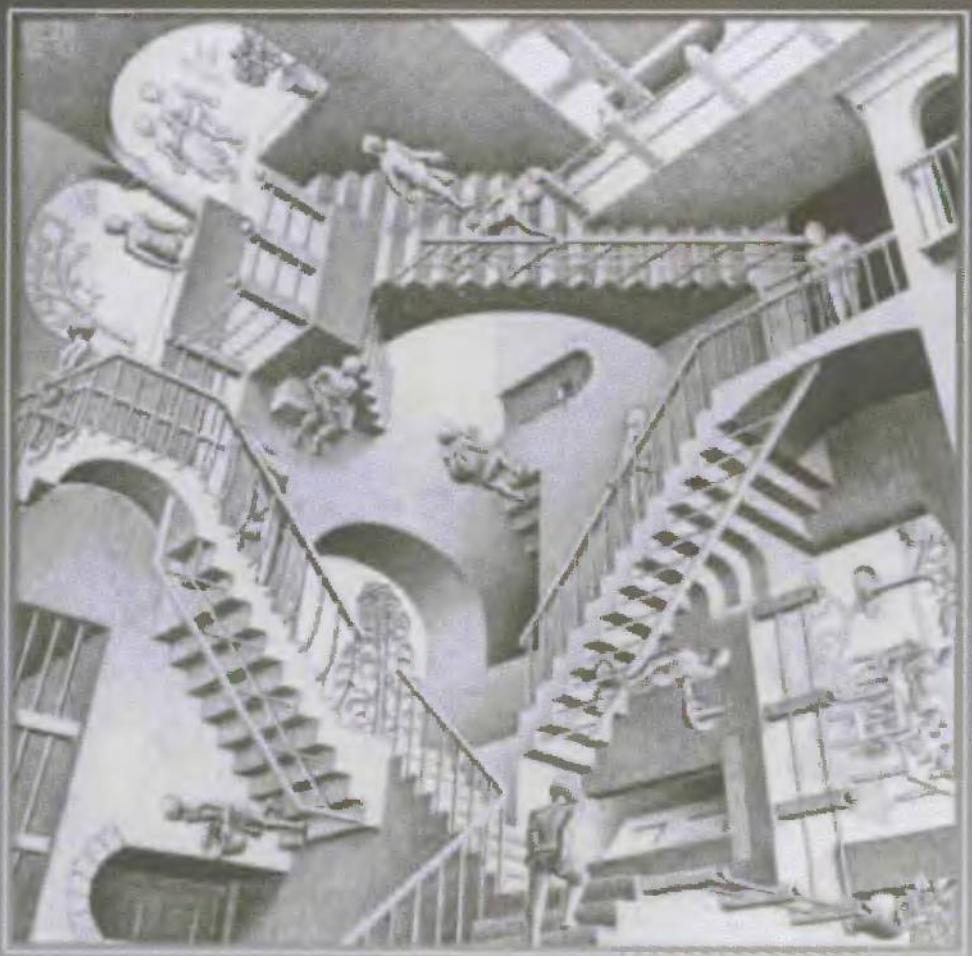
KNAPSACK PROBLEMS

Algorithms and Computer Implementations

SILVANO MARTELLO · PAOLO TOTH



KNAPSACK PROBLEMS

Algorithms and Computer Implementations

Silvano Martello

and

Paolo Toth

DEIS, University of Bologna

JOHN WILEY & SONS

Chichester · New York · Brisbane · Toronto · Singapore

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Jacaranda Wiley Ltd, G.P.O. Box 859, Brisbane,
Queensland 4001, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W 1L1, Canada

John Wiley & Sons (SEA) Pte Ltd, 37 Jalan Pemimpin #05-04,
Block B, Union Industrial Building, Singapore 2057

Library of Congress Cataloging-in-Publication Data:
Martello, Silvano.
Knapsack problems : algorithms and computer implementations
Silvano Martello, Paolo Toth.
p. cm. — (Wiley-Interscience series in discrete mathematics
and optimization)
Includes bibliographical references.
ISBN 0 471 92420 2
1. Computational complexity. 2. Mathematical optimization.
3. Algorithms. 4. Linear programming. 5. Integer programming.
I. Toth, Paolo. II. Title. III. Series.
QA 267.7.M37 1990
511′.8—dc20 90-12279
CIP

British Library Cataloguing in Publication Data:
Martello, Silvano
Knapsack problems : algorithms and computer
implementations.
1. Linear programming. Computation
I. Title II. Toth, Paolo
519.72

ISBN 0 471 92420 2

Printed in Great Britain by Biddles Ltd, Guildford

Contents

Preface

The development of computational complexity theory has led, in the last fifteen years, to a fascinating insight into the inherent difficulty of combinatorial optimization problems, but has also produced an undesirable side effect which can be summarized by the "equation"

$$NP\text{-}hardness \;=\; intractability,$$

thereby diminishing attention to the study of exact algorithms for NP-hard problems. However, recent results on the solution of very large instances of integer linear programming problems with special structure on the one hand, and forty years of successful use of the simplex algorithm on the other, indicate the concrete possibility of solving problems exactly through worst-case exponential-time algorithms.

This book presents a state-of-art on exact and approximate algorithms for a number of important NP-hard problems in the field of integer linear programming, which we group under the term *knapsack*. The choice of the problems reflects our personal involvement in the field, through a series of investigations over the past ten years. Hence the reader will find not only the "classical" knapsack problems (*binary, bounded, unbounded, binary multiple*), but also less familiar problems (*subset-sum, change-making*) or well-known problems which are not usually classified in the knapsack area (*generalized assignment, bin-packing*). He will find no mention, instead, of other knapsack problems (fractional, multidimensional, non-linear), and only a limited treatment of the case of generalized upper bound constraints.

The goal of the book is to fully develop an algorithmic approach without losing mathematical rigour. For each problem, we start by giving a mathematical model, discussing its relaxations and deriving procedures for the computation of bounds. We then develop approximate algorithms, approximation schemes, dynamic programming techniques and branch-and-bound algorithms. We analyse the computational complexity and the worst-case performance of bounds and approximate methods. The average performance of the computer implementations of exact and approximate algorithms is finally examined through extensive computational experiments. The Fortran codes implementing the most effective methods are provided in the included diskette. The codes are portable on virtually any computer, extensively commented and—hopefully—easy to use.

For these reasons, the book should be appreciated both by academic researchers

and industrial practitioners. It should also be suitable for use as a supplementary text in courses emphasizing the theory and practice of algorithms, at the graduate or advanced undergraduate level. The exposition is in fact self-contained and is designed to introduce the reader to a methodology for developing the link between mathematical formulation and effective solution of a combinatorial optimization problem. The simpler algorithms introduced in the first chapters are in general extensively described, with numerous details on the techniques and data structures used, while the more complex algorithms of the following chapters are presented at a higher level, emphasizing the general philosophy of the different approaches. Many numerical examples are used to clarify the methodologies introduced. For the sake of clarity, all the algorithms are presented in the form of pseudo-Pascal procedures. We adopted a structured exposition for the polynomial and pseudo-polynomial procedures, but allowed a limited use of "go to" statements for the branch-and-bound algorithms. (Although this could, of course, have been avoided, the resulting exposition would, in our opinion, have been much less readable.)

We are indebted to many people who have helped us in preparing this book. Jan Karel Lenstra suggested the subject, and provided guidance and encouragement during the two years of preparation. Mauro Dell'Amico, Laureano Escudero and Matteo Fischetti read earlier versions of the manuscript, providing valuable suggestions and pointing out several errors. (We obviously retain the sole responsibility for the surviving errors.) Constructive comments were also made by Egon Balas, Martin Dyer, Ronald Graham, Peter Hammer, Ben Lageweg, Gilbert Laporte, Manfred Padberg, David Shmoys, Carlo Vercellis and Laurence Wolsey. The computational experiments and computer typesetting with the TEX system were carried out by our students Andrea Bianchini, Giovanna Favero, Marco Girardini, Stefano Gotra, Nicola Moretti, Paolo Pinetti and Mario Zacchei.

We acknowledge the financial support of the Ministero della Pubblica Istruzione and the Consiglio Nazionale delle Ricerche. Special thanks are due to the Computing Centre of the Faculty of Engineering of the University of Bologna and its Director, Roberto Guidorzi, for the facilities provided in the computational testing of the codes.

Bologna, Italy
July 1989

SILVANO MARTELLO
PAOLO TOTH

1

Introduction

1.1 WHAT ARE KNAPSACK PROBLEMS?

Suppose a hitch-hiker has to fill up his knapsack by selecting from among various possible objects those which will give him maximum comfort. This *knapsack problem* can be mathematically formulated by numbering the objects from 1 to n and introducing a vector of binary variables x_j ($j = 1, \ldots, n$) having the following meaning:

$$x_j = \begin{cases} 1 & \text{if object } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

Then, if p_j is a measure of the comfort given by object j, w_j its size and c the size of the knapsack, our problem will be to select, from among all binary vectors x satisfying the *constraint*

$$\sum_{j=1}^{n} w_j x_j \leq c,$$

the one which *maximizes* the *objective function*

$$\sum_{j=1}^{n} p_j x_j.$$

If the reader of this book does not, or no longer practises hitch-hiking, he might be more interested in the following problem. Suppose you want to invest—all or in part—a capital of c dollars and you are considering n possible investments. Let p_j be the profit you expect from investment j, and w_j the amount of dollars it requires. It is self-evident that the optimal solution of the knapsack problem above will indicate the best possible choice of investments.

At this point you may be stimulated to solve the problem. A naive approach would be to program a computer to examine all possible binary vectors x, selecting the best of those which satisfy the constraint. Unfortunately, the number of such vectors is 2^n, so even a hypothetical computer, capable of examining one billion vectors per second, would require more than 30 years for $n = 60$, more than 60 years for $n = 61$, ten centuries for $n = 65$, and so on. However, specialized algorithms can, in most cases, solve a problem with $n = 100\,000$ in a few seconds on a mini-computer.

The problem considered so far is representative of a variety of knapsack-type problems in which a set of entities are given, each having an associated value and size, and it is desired to select one or more disjoint subsets so that the sum of the sizes in each subset does not exceed (or equals) a given bound and the sum of the selected values is maximized.

Knapsack problems have been intensively studied, especially in the last decade, attracting both theorists and practicians. The theoretical interest arises mainly from their simple structure which, on the one hand allows exploitation of a number of combinatorial properties and, on the other, more complex optimization problems to be solved through a series of knapsack-type subproblems. From the practical point of view, these problems can model many industrial situations: capital budgeting, cargo loading, cutting stock, to mention the most classical applications. In the following chapters we shall examine the most important knapsack problems, analysing relaxations and upper bounds, describing exact and approximate algorithms and evaluating their efficiency both theoretically and through computational experiments. The Fortran codes of the principal algorithms are provided in the floppy disk accompanying the book.

1.2 TERMINOLOGY

The objects considered in the previous section will generally be called *items* and their number be indicated by n. The value and size associated with the jth item will be called *profit* and *weight*, respectively, and denoted by p_j and w_j ($j = 1, \ldots, n$).

The problems considered in Chapters 2 to 5 are *single knapsack problems*, where one *container* (or *knapsack*) must be filled with an optimal subset of items. The *capacity* of such a container will be denoted by c. Chapters 6 to 8 deal with *multiple knapsack problems*, in which more than one container is available.

It is always assumed, as is usual in the literature, that profits, weights and capacities are positive integers. The results obtained, however, can easily be extended to the case of real values and, in the majority of cases, to that of nonpositive values.

The prototype problem of the previous section,

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

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

$$x_j = 0 \text{ or } 1, \qquad j = 1, \ldots, n,$$

is known as the *0-1 Knapsack Problem* and will be analysed in Chapter 2. In Section 2.12 we consider the generalization arising when the item set is partitioned into

subsets and the additional constraint is imposed that at most one item per subset is selected (*Multiple-Choice Knapsack Problem*).

The problem can be generalized by assuming that for each j ($j = 1, \ldots, n$), b_j items of profit p_j and weight w_j are available ($b_j \leq c/w_j$): thus we obtain the *Bounded Knapsack Problem*, defined by

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

The problem is considered in Chapter 3. The special case in which $b_j = +\infty$ for all j (*Unbounded Knapsack Problem*) is treated in Section 3.6.

In Chapter 4 we examine the particular case of the 0-1 knapsack problem arising when $p_j = w_j$ ($j = 1, \ldots, n$), as frequently occurs in practical applications. The problem is to find a subset of weights whose sum is closest to, without exceeding, the capacity, i.e.

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

generally referred to as the *Subset-Sum Problem*.

In Chapter 5 a very particular bounded knapsack problem is considered, arising when $p_j = 1$ ($j = 1, \ldots, n$) and, in the capacity constraint, we impose equality instead of inequality. This gives

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j = c, \\ & 0 \leq x_j \leq b_j \qquad j = 1, \ldots, n, \\ & x_j \text{ integer} \qquad j = 1, \ldots, n, \end{aligned}$$

usually called the *Change-Making Problem*, since it recalls the situation of a cashier having to assemble a given change c using the maximum (or minimum) number of coins. The same chapter deeply analyses the *Unbounded Change-Making Problem*, in which $b_j = +\infty$ for all j.

An important generalization of the 0-1 knapsack problem, discussed in Chapter 6, is the *0-1 Multiple Knapsack Problem*, arising when m containers, of given capacities c_i $(i = 1, \ldots, m)$ are available. By introducing binary variables x_{ij}, taking value 1 if item j is selected for container i, 0 otherwise, we obtain the formulation

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

Now consider a 0-1 multiple knapsack problem in which the profit and weight of each item vary according to the container for which they are selected. By defining p_{ij} (resp. w_{ij}) as the profit (resp. the weight) of item j if inserted in container i, we get

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j=1}^{n} w_{ij} x_{ij} \leq c_i, \qquad i = 1, \ldots, m, \\
& \sum_{i=1}^{m} x_{ij} \leq 1, \qquad j = 1, \ldots, n, \\
& x_{ij} = 0 \text{ or } 1, \qquad i = 1, \ldots, m, j = 1, \ldots, n,
\end{aligned}$$

known as the *Generalized Assignment Problem*, which is dealt with in Chapter 7. This is not, strictly speaking, a knapsack problem, but is included in this review because knapsack subproblems play a central role in the algorithms for solving it.

The problem is generally viewed as that of optimally assigning, all or in part, n jobs to m machines (n tasks to m agents, and so on), given the profit, p_{ij}, obtainable if machine i is assigned job j, the corresponding resource, w_{ij}, required, and the amount, c_i, of resource available to machine i.

In Chapter 8 we consider the well-known *Bin-Packing Problem*, which is not usually included in the knapsack area, but can be interpreted as a multiple subset-sum problem where all containers have the same capacity c, all items must be selected and it is desired to minimize the number of containers used. Given any upper bound m on the number of containers, and introducing m binary variables y_i, taking value 0 if container i is used, value 1 otherwise, we can state the problem as:

$$\text{maximize} \quad \sum_{i=1}^{m} y_i$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_{ij} \leq c(1 - y_i), \qquad i = 1, \ldots, m,$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, \ldots, n,$$

$$y_i = 0 \text{ or } 1, \qquad i = 1, \ldots, m,$$

$$x_{ij} = 0 \text{ or } 1, \qquad i = 1, \ldots, m, j = 1, \ldots, n.$$

In the last decades, an impressive amount of research on knapsack problems has been published in the literature. Reviews have been presented in the following surveys:

Salkin and De Kluyver (1975) present a number of industrial applications and results in transforming integer linear programs to knapsack problems (an approach which appeared very promising at that time);

Martello and Toth (1979) consider exact algorithms for the zero-one knapsack problem and their average computational performance; the study is extended to the other linear knapsack problems and to approximate algorithms in Martello and Toth (1987);

Dudzinski and Walukiewicz (1987) analyse dual methods for solving Lagrangian and linear programming relaxations.

In addition, almost all books on integer programming contain a section on knapsack problems. Mention is made of those by Hu (1969), Garfinkel and Nemhauser (1972), Salkin (1975), Taha (1975), Papadimitriou and Steiglitz (1982), Syslo, Deo and Kowalik (1983), Schrijver (1986), Nemhauser and Wolsey (1988).

1.3 COMPUTATIONAL COMPLEXITY

We have so far introduced the following problems:

0-1 KNAPSACK;
BOUNDED KNAPSACK;
SUBSET-SUM;
CHANGE-MAKING;
0-1 MULTIPLE KNAPSACK;
GENERALIZED ASSIGNMENT;
BIN-PACKING.

We will now show that all these problems are *NP-hard* (we refer the reader to Garey and Johnson (1979) for a thorough discussion on this concept). For each problem P, we either prove that its *recognition version* R(P) is NP-complete or that it is a generalization of a problem already proved to be NP-hard.

The following recognition problem:

PARTITION: given n positive integers $w_1, \ldots, w_n$, is there a subset $S \subseteq N = \{1, \ldots, n\}$ such that $\sum_{j \in S} w_j = \sum_{j \in N \setminus S} w_j$?

is a basic NP-complete problem, originally treated in Karp (1972).

(a) *SUBSET-SUM is NP-hard.*

Proof. Consider R(SUBSET-SUM), i.e.: given $n+2$ positive integers $w_1, \ldots, w_n, c$ and a, is there a subset $S \subseteq N = \{1 \ldots . n\}$ such that $\sum_{j \in S} w_j \leq c$ and $\sum_{j \in S} w_j \geq a$?

Any instance I of PARTITION can be *polynomially transformed* into an equivalent instance I' of R(SUBSET-SUM) by setting $c = a = \sum_{j \in N} w_j/2$ (the answer for I is "yes" if and only if the answer for I' is "yes"). □

(b) *0-1 KNAPSACK is NP-hard.*

Proof. SUBSET-SUM is the particular case of 0-1 KNAPSACK when $p_j = w_j$ for all $j \in N$. □

(c) *BOUNDED KNAPSACK is NP-hard.*

Proof. 0-1 KNAPSACK is the particular case of BOUNDED KNAPSACK when $b_j = 1$ for all $j \in N$. □

(d) *CHANGE-MAKING is NP-hard.*

Proof. We prove NP-hardness of the special case in which $b_j = 1$ for all j. Consider R(CHANGE-MAKING), i.e.: given $n + 2$ positive integers $w_1, \ldots, w_n$, c and a, is there a subset $S \subseteq N = \{1, \ldots, n\}$ such that $\sum_{j \in S} w_j = c$ and $|S| \geq a$? Any instance I of PARTITION can be polynomially transformed into an equivalent instance I' of R(CHANGE-MAKING) by setting $c = \sum_{j \in N} w_j/2$ and $a = 1$. □

Consequently, these single knapsack problems cannot be solved in a time bounded by a polynomial in n, unless $\mathcal{P} = \mathcal{NP}$. All of them, however, admit a *pseudo-polynomial* algorithm, i.e. an algorithm whose time (and space) complexity is bounded by a polynomial in n and c. In fact, it can easily be verified that the following dynamic programming recursions solve the corresponding problems. (More detailed descriptions can be found in the specific chapters.) Given any instance of a single knapsack problem, consider the sub-instance defined by items $1, \ldots, j$ and capacity u ($j \leq n$, $u \leq c$). Let $f_j(u)$ be the corresponding optimal solution value ($f_j(u) = -\infty$ if no feasible solution exists) and $S_j(u)$ the optimal subset of items. The optimal solution value of the problem, $f_n(c)$, can then be obtained by iteratively applying the following recursive formulae:

0-1 KNAPSACK:

$$f_1(u) = \begin{cases} 0 & \text{for } u = 0, \ldots, w_1 - 1; \\ p_1 & \text{for } u = w_1, \ldots, c; \end{cases}$$

$$f_j(u) = \max(f_{j-1}(u), f_{j-1}(u - w_j) + p_j) \text{ for } j = 2, \ldots, n \text{ and } u = 0, \ldots, c;$$

time complexity $O(nc)$.

BOUNDED KNAPSACK:

$$f_1(u) = \begin{cases} lp_1 & \text{for } l = 0, \ldots, b_1 - 1 \text{ and } u = lw_1, \ldots, (l+1)w_1 - 1; \\ b_1 p_1 & \text{for } u = b_1 w_1, \ldots, c; \end{cases}$$

$$f_j(u) = \max\{f_{j-1}(u - lw_j) + lp_j : 0 \leq l \leq b_j\} \text{ for } j = 2, \ldots, n \text{ and } u = 0, \ldots, c;$$

time complexity $O(c \sum_{j=1}^{n} b_j)$, that is, in the worst case, $O(nc^2)$.

SUBSET-SUM:

Same as 0-1 KNAPSACK, but with p_j replaced by w_j.

CHANGE-MAKING:

$$f_1(u) = \begin{cases} l & \text{for } u = lw_1, \text{ with } l = 0, \ldots, b_1; \\ -\infty & \text{for all positive } u \leq c \text{ such that } u(\text{mod } w_1) \neq 0 ; \end{cases}$$

$$f_j(u) = \max\{ f_{j-1}(u - lw_j) + l : \quad 0 \leq l \leq b_j \} \text{ for } j = 2, \ldots, n \text{ and } u = 0, \ldots, c;$$

time complexity $O(c \sum_{j=1}^{n} b_j)$, that is, in the worst case, $O(nc^2)$.

For all the algorithms the computation of $S_j(u)$ is straightforward. Since, for each j, we only need to store $S_{j-1}(u)$ and $S_j(u)$ for all u, the space complexity is always $O(nc)$.

For the multiple problems (0-1 MULTIPLE KNAPSACK, GENERALIZED ASSIGNMENT, BIN-PACKING) no pseudo-polynomial algorithm can exist, unless $\mathcal{P} = \mathcal{NP}$, since the problems can be proved to be *NP-hard in the strong sense*. Consider in fact the following recognition problem:

3-PARTITION: given $n = 3m$ positive integers $w_1 \ldots . w_n$ satisfying $\sum_{j=1}^{n} w_j/m = B$ integer and $B/4 < w_j < B/2$ for $j = 1, \ldots, n$, is there a partition of $N = \{1, \ldots, n\}$ into m subsets $S_1, \ldots, S_m$ such that $\sum_{j \in S_i} w_j = B$ for $i = 1, \ldots, m$? (Notice that each S_i must contain exactly three elements from N.)

This is the first problem discovered to be NP-complete in the strong sense (Garey and Johnson, 1975).

(e) *0-1 MULTIPLE KNAPSACK is NP-hard in the strong sense.*

Proof. Consider R(0-1 MULTIPLE KNAPSACK), i.e.: given $2n + m + 1$ positive integers: $p_1, \ldots, p_n$; $w_1, \ldots, w_n$; $c_1, \ldots, c_m$, and a, are there m disjoint subsets $S_1, \ldots, S_m$ of $N = \{1, \ldots, n\}$ such that $\sum_{j \in S_i} w_j \leq c_i$ for $i = 1, \ldots, m$ and $\sum_{i=1}^{m} \sum_{j \in S_i} p_j \geq a$? Any instance I of 3-PARTITION can be *pseudo-polynomially transformed* into an equivalent instance I' of R(0-1 MULTIPLE KNAPSACK) by setting $c_i = B$ for $i = 1, \ldots, m$, $p_j = 1$ for $j = 1, \ldots, n$ and $a = n$ (which implies that $\bigcup_{i=1}^{m} S_i = N$ in any "yes" instance). □

(f) *GENERALIZED ASSIGNMENT is NP-hard in the strong sense.*

Proof. Immediate, since 0-1 MULTIPLE KNAPSACK is the particular case of GENERALIZED ASSIGNMENT when $p_{ij} = p_j$ and $w_{ij} = w_j$ for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. □

(g) *BIN-PACKING is NP-hard in the strong sense.*

Proof. Consider R(BIN-PACKING), i.e.: given $n+2$ positive integers $w_1, \ldots, w_n$, c and a, is there a partition of $N = \{1, \ldots, n\}$ into a subsets $S_1, \ldots, S_a$ such that $\sum_{j \in S_i} w_j \leq c$ for $i = 1, \ldots, a$? Any instance I of 3-PARTITION can be pseudo-polynomially transformed into an equivalent instance I' of R(BIN-PACKING) by setting $c = B$ and $a = m$. □

1.4 LOWER AND UPPER BOUNDS

In the previous section we have proved that none of our problems can be solved in polynomial time, unless $\mathcal{P} = \mathcal{NP}$. Hence in the following chapters we analyse:

(a) enumerative algorithms (having, in the worst case, running times which grow exponentially with the input size) to determine optimal solutions;

(b) approximate algorithms (with running times bounded by a polynomial in the input size) to determine feasible solutions whose value is a *lower bound* on the optimal solution value.

The average running times of such algorithms are experimentally evaluated through execution of the corresponding computer codes on different classes of randomly-generated test problems. It will be seen that the average behaviour of the enumerative algorithms is in many cases much better than the worst-case bound, allowing optimal solution of large-size problems with acceptable running times.

The *performance* of an approximate algorithm for a specific instance is measured through the ratio between the solution value found by the algorithm and the optimal solution value (notice that, for a maximization problem, this ratio is no greater than one). Besides the experimental evaluation, it is useful to provide, when possible, a theoretical measure of performance through *worst-case* analysis (see Fisher (1980) for a general introduction to this concept).

Let A be an approximate algorithm for a given maximization problem (all our considerations extend easily to the minimization case). For any instance I of the problem, let $OPT(I)$ be the optimal solution value and $A(I)$ the value found by A. Then, the *worst-case performance ratio* of A is defined as the largest real number $r(A)$ such that

$$\frac{A(I)}{OPT(I)} \geq r(A) \qquad \text{for all instances } I;$$

the closer $r(A)$ is to one, the better the worst-case behaviour of A. The proof that a given value r is the worst-case performance ratio of an algorithm A consists, in general, of two phases:

(i) it is first proved that, for any instance I of the problem, inequality $A(I)/OPT(I) \geq r$ holds;

(ii) in order to ensure that r is the largest value satisfying the inequality, i.e. that r is *tight*, a specific instance I' is produced for which $A(I')/OPT(I') = r$ holds (or a series of instances for which the above ratio tends to be arbitrarily close to r).

The performance of A can be equivalently expressed in terms of *worst-case relative error*, i.e. the smallest real number $\varepsilon(A)$ such that

$$\frac{OPT(I) - A(I)}{OPT(I)} \leq \varepsilon(A) \qquad \text{for all instances } I.$$

(i.e. $r(A) = 1 - \varepsilon(A)$).

An *approximation scheme* for a maximization problem is an algorithm A which, given an instance I and an error bound $\varepsilon > 0$, returns a solution of value $A(I)$ such that $(OPT(I) - A(I))/OPT(I) \leq \varepsilon$. Let *length* (I) denote the *input size*, i.e. the number of symbols required for coding I. If, for any fixed ε, the running time of A is bounded by a polynomial in *length* (I), then A is a *polynomial-time approximation scheme*: any relative error can be obtained in a time which is polynomial in *length* (I) (but can be exponential in $1/\varepsilon$). If the running time of A is polynomial both in *length* (I) and $1/\varepsilon$, then A is a *fully polynomial-time approximation scheme*.

In subsequent chapters we describe the most interesting polynomial-time and fully polynomial-time approximation schemes for single knapsack problems. For the remaining (multiple) problems, no fully polynomial-time approximation scheme can exist, unless $\mathcal{P} = \mathcal{NP}$, since (see Garey and Johnson (1975)) this would imply the existence of a pseudo-polynomial algorithm for their optimal solution (which is impossible, these being NP-hard problems in the strong sense). For BIN-PACKING, also the existence of a polynomial-time approximation scheme can be ruled out, unless $\mathcal{P} = \mathcal{NP}$ (Johnson, Demers, Ullman, Garey and Graham, 1974). The same holds for GENERALIZED ASSIGNMENT and 0-1 MULTIPLE KNAPSACK in the minimization version (Sahni and Gonzalez, 1976). For the maximization version of these two problems no polynomial-time approximation scheme is known, although there is no proof that it cannot exist (the proof in Sahni and Gonzalez (1976) does not extend to the maximization case).

Besides experimental and worst-case analysis, an approximate algorithm can allow *probabilistic analysis*. Speaking informally this consists of specifying an average problem instance in terms of a probability distribution over the class of all instances and evaluating running time and solution value as random variables. Examples of this approach which, however, is generally possible only for very simple algorithms, are given in Sections 2.8.3 and 4.3.4 (see Karp, Lenstra, McDiarmid and Rinnooy Kan (1985) and Rinnooy Kan (1987) for a general introduction to probabilistic analysis).

For a maximization problem, the solution value determined by an approximate algorithm limits the optimal solution value from below. It is always convenient to

have methods for limiting this value from above, too. *Upper bounds* are extremely useful

(a) in enumerative algorithms, to exclude computations which cannot lead to the optimal solution;

(b) in approximate algorithms, to "a-posteriori" evaluate the performance obtained. Suppose algorithm A is applied to instance I, and let $U(I)$ be any upper bound on $OPT(I)$: it is then clear that the relative error of the approximate solution is no greater than $(U(I) - A(I))/U(I)$.

The worst-case performance ratio of an upper bounding procedure U can be defined similarly to that of an approximate algorithm, i.e. as the smallest real number $\rho(U)$ such that

$$\frac{U(I)}{OPT(I)} \leq \rho(U) \qquad \text{for all instances } I.$$

The closer $\rho(U)$ is to one, the better the worst-case behaviour of U.

Upper bounds are usually computed by solving *relaxations* of the given problems. *Continuous, Lagrangian* and *surrogate* relaxations are the most frequently used. For a given problem P, the corresponding relaxed problem will be denoted with $C(P)$, $L(P, m)$ and $S(P, m)$, m being an appropriate vector of *multipliers*. The optimal solution value of problem P will be denoted with $z(P)$.