School of Mathematics and Statistics

**Algorithms
for
Some Hard Knapsack Problems**

Araya Kulanoot

This thesis is presented as part of the requirements for
the award of the Degree of Doctor of Philosophy
of the
Curtin University of Technology

January 2000

Summary

The Knapsack Problems are among the simplest integer programs which are *NP-hard*. Problems in this class are typically concerned with selecting from a set of given items, each with a specified weight and value, a subset of items whose weight sum does not exceed a prescribed capacity and whose value is maximum. The specific problem that arises depends on the number of knapsacks (single or multiple) to be filled and on the number of available items of each type (bounded or unbounded). The classical 0-1Knapsack Problem arises when there is one knapsack and one item of each type.

Knapsack Problems have been intensively studied over the past forty years because of their direct application to problems arising in industry (for example, cargo loading, cutting stock, and budget control) and also for their contribution to the solution methods for integer programming problems.

Several exact algorithms based on branch and bound and dynamic programming have been proposed to solve the Knapsack Problems. For some types of data instances, very efficient algorithms have been found. However, a number of hard knapsack instances have been identified. For example, subset sum and strongly correlated data types. This has motivated some researchers to develop specialized algorithms for particular hard problems.

This thesis considers a number of hard Knapsack Problems with a single constraint. A number of specialized algorithms are developed. Our work focuses on some hard instances of the 0-1Knapsack Problem, the *Bounded Knapsack Problem*, the *Unbounded Knapsack Problem* and the *Change-Making Problem*.

Chapter 1 provides the background of the Knapsack Problem including some important Knapsack Problems and standard types of data instances, terminology, and a summary of our work. **Chapter 2** gives a review of the applications and the solution methods that have been proposed. The remaining chapters present the details of our specialized algorithms.

Chapter 3 proposes algorithms for the hard 0-1Knapsack Problems instances of subset sum, strongly correlated, and inverse strongly correlated. The algorithms for the Bounded Knapsack Problem instances of strongly correlated and subset sum are also presented. Extensive computational results show that our algorithms are able to solve large problems of size up to one million variables in less than 7 seconds.

Chapter 4 proposes algorithms for some hard Unbounded Knapsack Problems. Two algorithms one for the Unbounded Strongly Correlated Knapsack Problem (algorithm CKU1) and one for the Unbounded Subset Sum Problem (algorithm CKU2) are presented. Extensive computational results establish that our two algorithms are able to solve large problems with up to one million variables in less than 0.3 second.

Finally, **Chapter 5** proposes exact algorithms for the Change-Making Problem. The problem is a particular type of single Knapsack Problems. This chapter proposes two exact algorithms: algorithm CKUC for the Unbounded Change-Making Problem (UCMP) and algorithm CKBC for the Bounded Change-Making Problem (BCMP). The algorithms can solve large-sized problems, when the item types are generated in small ranges, in less than 51 milliseconds for UCMP and less than 3.5 seconds for BCMP.

Certification

I certify that the work presented in this thesis is my own work and that all references are duly acknowledged. This work has not been submitted, in whole or in part, in respect of any academic award at Curtin University of Technology or elsewhere.

Araya Kulanoot

January 2000

Acknowledgements

I am most grateful to my thesis supervisor, Professor Lou Caccetta, for his guidance, support, and encouragement. His helpful suggestions have been invaluable in the preparation of this thesis.

I am very indebted to Professor David Pisinger for his excellent development of the Knapsack algorithms, which has been a great inspiration for my project. I would like to thank him for providing me some of his algorithm codes.

Special thanks and appreciation go to my parents and my sister for their love, support and encouragement throughout my entire education. I express my appreciation to Dr. Narong Punnim for his continuous support and encouragement during my Ph.D. study.

I thank my friend, Yohaness Handra, a former postgraduate student at School of Computing, Curtin University, for his help in some of programming. Also thanks to Dr. Roger Collinson for his helpful suggestion in a programming code.

Finally, I am grateful to the Royal Thai Government for providing me with a Ph.D. scholarship. I would also like to thank Thammasat University and all of my colleagues at Department of Mathematics and Statistics for giving me a period of time to undertake this study.

Araya Kulanoot
Perth, Western Australia.

Table of contents

Summary	i
Certification	iii
Acknowledgements	iv
Chapter 1 Introduction	1
1.1 Background	1
1.2 Terminology	10
1.3 Review and Outline of the Thesis	12
Chapter 2 Literature Review	17
2.1 Applications	18
2.1.1 Reduction of Integer linear programming to Knapsack Problem	18
2.1.2 Upper bound for some Integer programming problems	20
2.1.3 Cutting Plane Generation	24
2.1.4 Penalty estimation of Integer programming	26
2.2 Connection with Networks	27
2.3 Some Properties of the 0-1Knapsack Problem	29
2.4 Upper Bound Improvements	32
2.4.1 Bounds from Additional Constraints of some variables	33
2.4.2 Bounds from Lagrangian Relaxation	34

2.4.3	Bounds from Partial Enumeration	37
2.4.4	Bounds from Valid Inequalities	37
2.5	Reduction Procedures	41
2.6	Core Problem in Knapsack Algorithms	45
2.7	Exact Algorithms for the 0-1 Knapsack Problem	46
2.7.1	Branch and Bound Algorithms	46
2.7.2	Dynamic Programming Algorithms	51
2.7.3	Approximation Algorithms	54

Chapter 3 Algorithms for Some Hard Knapsack

Problems	57
3.1 Introduction	58
3.2 Preliminaries	62
3.3 Subset Sum Problem (SSP)	68
3.3.1 Computational Results	72
3.4 Strongly Correlated Problem (SCKP)	75
3.4.1 Computational Results	78
3.5 Inverse Strongly Correlated Knapsack Problem (ISCKP)	81
3.5.1 Computational Results	83
3.6 Bounded Knapsack Problem (BKP)	85
3.6.1 Computational Results	90
3.7 Conclusions	94

Chapter 4 Algorithms for Some Hard Unbounded

Knapsack Problems	96
4.1 Introduction	97
4.2 Upper Bound	102
4.3 Unbounded Strongly Correlated Knapsack Problem (UNSCK)	105
4.3.1 Dominance Relation	106
4.3.2 Algorithm CKU1 for UNSCK	107

4.3.3 Computational Results	118
4.3.4 Hard Instances	122
4.4 Unbounded Subset Sum Problem (UNSSP)	124
4.4.1 Computational Results	126
4.5 Conclusions	128
 Chapter 5 Exact Algorithms for Change-Making	
Problems	130
5.1 Introduction and background	131
5.2 Algorithm NEXCOM	135
5.3 Unbounded Change-Making Problem (UCMP)	136
5.3.1 Lower Bounds	136
5.3.2 Algorithm CKUC	138
5.3.3 Computational Results	150
5.4 Bounded Change-Making Problem (BCMP)	154
5.4.1 Algorithm CKBC	155
5.4.2 Computational Results	161
5.5 Conclusions	166
 Chapter 6 Conclusions	167
 References	171

CHAPTER 1

Introduction

1.1 Background

The Knapsack Problem is one of the important integer linear programming problems that has been extensively studied over the past forty years. The problem hypothetically arises as the situation of a hiker having to fill up his/her knapsack by selecting from among various possible items each of which has a specified weight and value. The capacity of the knapsack is limited and the hiker would like to maximize the total value of the items in the knapsack. The classical Knapsack Problem is to select a subset of items such that the total value is maximized and the total weight does not exceed the knapsack capacity.

Mathematically, given a knapsack with a capacity of c units and a set of n items labelled $1, 2, \dots, n$, with item i having a weight of w_i units and a value of p_i , the problem may be stated as :

$$\begin{aligned}
& \text{maximize} && \sum_{i=1}^n p_i x_i \\
& \text{subject to} && \sum_{i=1}^n w_i x_i \leq c, \\
& && x_i = 0 \text{ or } 1, i = 1, 2, \dots, n,
\end{aligned}$$

where the variable $x_i = 1$ if item i is chosen and 0 otherwise.

The above Knapsack Problem with binary variables is called the classical **0-1 Knapsack Problem (0-1KP)**. This problem arises naturally in business and industry. For example, in financial management where one needs to select from a set of n projects each with a prescribed cost (w_i) and return (p_i) a subset which maximizes the total return and which does not violate the available funds (c). Over the past forty years the 0-1 Knapsack Problem together with numerous variations (as discussed below) has been studied, giving rise to what is sometimes referred to as the “*Family of Knapsack Problems*”. Practical and theoretical reasons have motivated interest in this family of problems.

Knapsack Problems arise in many business and industrial applications including: financial management (Cord, 1964; Kaplan, 1966; Hansmann, 1961); cargo loading (Bellman & Dreyfus 1962, pp.27-31); cutting stock (Gilmore & Gomory, 1963,1966); fault tolerance (Sinha & Zoltners, 1979); and cryptography (Diffie & Hellman, 1976).

Theoretical interest in the Knapsack Problems is two fold. Firstly, there are many applications where a general integer programming problem can be transformed into a Knapsack Problem (Salkin 1975, pp.389-402; Syslo, Deo & Kowalik 1983, pp.176-211). Secondly, in the solution of many integer programming problems, the Knapsack Problem arises as a subproblem. This is the case, for example, in the set-partitioning problem (Syslo, Deo & Kowalik 1983, pp.194-197), and in generating cutting planes for some large-scale 0-1 integer programming problems (Balas 1975; Nemhauser & Wolsey 1980, pp.450-464).

We begin our work by detailing some of the different types of Knapsack Problems that have been studied in the literature.

The Bounded Knapsack Problem (BKP)

The BKP may be viewed as a generalization of the 0-1KP in which there are up to b_i items of type i that can be put in the knapsack. The problem is then defined as:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^n p_i x_i \\ &\text{subject to} && \sum_{i=1}^n w_i x_i \leq c, \\ &&& x_i \in \{0, 1, 2, \dots, b_i\}, i = 1, 2, \dots, n. \end{aligned}$$

Here $x_i \in \{0, 1, 2, \dots, b_i\}$ denotes the number of item type i selected in the knapsack.

The Unbounded Knapsack Problem (UKP)

The UKP is a particular case of the BKP since an unlimited number of each item type is available. Mathematically, the problem may be formulated as:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^n p_i x_i \\ &\text{subject to} && \sum_{i=1}^n w_i x_i \leq c, \\ &&& x_i \geq 0 \text{ and integer, } i = 1, 2, \dots, n. \end{aligned}$$

This problem can actually be viewed as a BKP with $b_i = \lfloor c / w_i \rfloor$.

The UKP often arises in cargo loading and packing (Bellman & Dreyfus 1962, pp.27-31). Suppose we need to load a vessel of capacity c with a cargo composed of different types of items with each item of type i having weight w_i and value p_i . The problem is how to load the vessel with the most valuable cargo. If the number of each item type i is limited as b_i , the problem is then the BKP.

The Change-Making Problem (CMP)

The CMP may arise as follows: a cashier has to make an amount of change c by using the least number of coins of different denominations $w_1, w_2, \dots, w_n$. The CMP is a very particular single Knapsack Problem in minimization form where the capacity constraint is imposed as an equality and $p_i = 1, i = 1, 2, \dots, n$. We can state the problem as:

$$\begin{aligned} & \text{minimize} && \sum_{i=1}^n x_i \\ & \text{subject to} && \sum_{i=1}^n w_i x_i = c, \\ & && x_i \geq 0 \text{ and integer, } i = 1, 2, \dots, n. \end{aligned}$$

This problem is called the *Unbounded Change-Making Problem* since an unlimited number of items for each type is available. The generalization of CMP, called the *Bounded Change-Making Problem*, is obtained by limiting the value of x_i as $0 \leq x_i \leq b_i$, where b_i is the upper bound on the availability of items of type i , $i = 1, 2, \dots, n$.

The Multiple-Choice Knapsack Problem (MCKP)

Given disjoint classes $N_i, i = 1, \dots, k$ of items, the MCKP is a 0-1KP such that exactly one item is chosen from a class N_i . The problem may be stated as:

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^k \sum_{j \in N_i} p_{ij} x_{ij} \\ & \text{subject to} && \sum_{i=1}^k \sum_{j \in N_i} w_{ij} x_{ij} \leq c, \\ & && \sum_{j \in N_i} x_{ij} = 1, \quad i = 1, \dots, k, \\ & && x_{ij} = 0 \text{ or } 1, \quad i = 1, \dots, k, j \in N_i, \end{aligned}$$

where p_{ij} and w_{ij} are the profit and weight of item j in class i . Here the binary variable x_{ij} is 1 if item j in class i was selected, and 0 otherwise. The constraint $\sum_{j \in N_i} x_{ij} = 1, i = 1, \dots, k$ restricts the choice to only one item from each class.

One of the applications of MCKP arises in menu planning (identified by Balintfy *et al.*, 1978) for determining what food items (dishes) should be selected from k menu courses. Suppose course i contains N_i dishes, $i = 1, 2, \dots, k$. Here w_{ij} and p_{ij} could, respectively, correspond to the nutritional value and preference rating of dish j ($j = 1, 2, \dots, N_i$) in course i ($i = 1, 2, \dots, k$). The problem is to maximize an individual's food preference subject to a nutritional constraint limited as an amount c and the constraint that only 1 dish from each course can be selected.

The 0-1 Multiple Knapsack Problem (MKP)

There are m knapsacks of different capacity $c_i, i = 1, 2, \dots, m$. The MKP is to select a subset of n item types to fill those knapsacks with only one item of each type such that the largest profit sum is obtained. The problem, named the 0-1 Multiple Knapsack Problem, may be formulated as:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^m \sum_{j=1}^n p_{ij} x_{ij} \\ &\text{subject to} && \sum_{j=1}^n w_j x_{ij} \leq c_i, \quad i = 1, \dots, m, \\ &&& \sum_{i=1}^m x_{ij} \leq 1, \quad j = 1, \dots, n, \\ &&& x_{ij} = 0 \text{ or } 1, \text{ for all } i \text{ and } j, \end{aligned}$$

$$\text{where} \quad x_{ij} = \begin{cases} 1 & , \text{if item } j \text{ is chosen for knapsack } i \\ 0 & , \text{otherwise.} \end{cases}$$

Martello & Toth (1990a, pp.167-168) give an example of an application in loading m liquids, which cannot be mixed, into n tanks.

The Generalized Assignment Problem (GAP)

Here, a MKP is considered under the conditions where the profit and weight of each item vary according to which knapsack the item is placed. The objective is to maximize profit and the constraints are the knapsack capacities. Given n items and m knapsacks $K_1, K_2, \dots, K_m$ with K_i having capacity c_i . Letting p_{ij} and w_{ij} be the *profit* and *weight* associated with assigning item j to knapsack i , $j = 1, 2, \dots, n$, the problem can be stated as:

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

$$\text{where} \quad x_{ij} = \begin{cases} 1 & , \text{if item } j \text{ is assigned to knapsack } i \\ 0 & , \text{otherwise.} \end{cases}$$

A well-known example of the GAP arises in the context of optimally assigning m machines to n tasks or jobs. Here p_{ij} denotes the profit and w_{ij} the resource required from assigning task j to machine i . Sometimes the problem is stated as a minimum cost problem. The objective is to minimize the total cost of the assignment. Let q_{ij} be the cost of item j assigned to knapsack i , then the objective function is

$$\text{minimize} \quad \sum_{i=1}^m \sum_{j=1}^n q_{ij} x_{ij} .$$

Several types of hybrid Knapsack Problem have also been studied by a number of researchers. The recent work on different hybrid Knapsack Problems may be summarized as:

Problems	Researchers
The collapsing Knapsack Problem	Fayard and Plateau (1994), Pfersch, Pisinger & Woeginger (1997)
The Nested Knapsack Problem	Dudzinski & Walukiewicz (1987)
The Nonlinear Knapsack Problem	Morin & Marten (1976), Hochbaum (1995)
The Inverse-parametric Knapsack Problem	Burkard & Pfersch (1994)
The Dynamic and Stochastic Knapsack problem	Kleywegt and Papastavrou (1998)
The setup Knapsack Problem	Chajakis & Guignard (1994)
The dichotomous Knapsack Problem	Gottlieb (1990)
The generalized Knapsack Problem with variables coefficients	Suzuki (1978) Holmberg & Jornsten (1996)
The knapsack sharing Problem	Brown (1979), Yamada & Futakawa (1997)
The max-min 0-1 Knapsack Problem	Yu (1996)

Table 2.1 Some hybrid Knapsack Problems

It is well known (see for example the books of Salkin (1975, pp.383-435) and Syslo, Deo & Kowalik (1983, pp.117-176)) that every integer program can be transformed into the one constraint Knapsack Problem. Classical knapsack problems such as those mentioned above have been very useful in many applications and are widely known. In many cases, when solving complex integer programming problems, a 0-1KP has to be solved repeatedly and quickly. For

example in the application of a fractional cutting-plane algorithm for 0-1 integer programs; see Nemhauser & Wolsey (1988, pp.459-464) for an illustration. From a practical point of view, (as mentioned previously) some Knapsack Problems are used to model many real-life problems such as capital budgeting, project selection, budget control and several loading problems. The algorithms for solving both one and two dimensional cutting stock problems by Gilmore and Gomory (1961,1963) are crucially dependent on how efficiently Knapsack Problems can be solved. Therefore, effective algorithms for the Knapsack Problem are useful for many integer programs.

Over the last several decades, extensive research has been carried out to solve several problems in the knapsack family, especially the one-constraint Knapsack Problem. Although Knapsack Problems have simple structures they have been proved to be computationally difficult (as *NP-hard* problems) by Garey & Johnson (1979). Numerous exact algorithms have been developed from the following principal approaches:

- **Dynamic Programming:** The method of dynamic programming is based on the idea that an optimal solution can be constructed from optimal solutions to subproblems. Sometimes a bounding test is incorporated into the dynamic programming to discard unpromising subproblems. Currently, most efficient algorithms for solving several types of Knapsack Problems are based on dynamic programming techniques.
- **Branch and Bound:** The method of branch and bound has been effectively used to tackle a number of computationally difficult problems. Basically, the idea is to subdivide (branch) the feasible solution set into successively smaller subsets, placing bounds on the objective function value over each subset, and using these bounds to discard certain subsets from further consideration. The success of the method is highly dependent on the branching rules, the search strategy and the quality of the lower and upper bounds.

Over the past 25 years, considerable advancement has been made in solving Knapsack Problems. This advancement takes advantage of improvements in optimization techniques and available computer hardware and software. The efficiency of algorithms can be measured by a complexity analysis and/or through the performance on test data sets. Certainly computational experimentation is necessary to establish the efficiency of an algorithm. For a comparative analysis, many different types of randomly generated instances have been considered, in the literature. One of the factors indicating the hardness of the Knapsack Problem is the relationship between profit and weight. In all test instances the weights are randomly distributed in a given range and the profits are determined as a function of the weights. Basically, different data types have been constructed based on correlation coefficient values (Aherns & Finke, 1975) which indicate the degree of the relationship between w_i and p_i , $i = 1, 2, \dots, n$. The standard data types for Knapsack Problems have been classified as follows (Pisinger & Toth, 1998):

- ***Uncorrelated data:*** Here, for each item there is no correlation between profit and weight. This means that the profit does not depend on the weight. This Uncorrelated data instance is generally easy to solve due to the large variation in the profit-to-weight ratio.
- ***Weakly correlated data:*** Here there is good correlation between the profit and the weight of each item. Typically, the profit is randomly generated for a range of different weights within approximately a few percent and thus the correlation is in fact quite high. Generally, weakly correlated data instances are not difficult to solve since the difference between profit and weight also makes a large difference of profit-to-weight ratios in a set of data.
- ***Strongly correlated data:*** In this type, the profit and weight differ by a positive constant. The strongly correlated data instance is the most difficult instance to solve. Most exact algorithms, based on the dynamic programming or the branch and bound approach, are inefficient in solving the problem for this data type due to the large gap between the continuous and integer solution

value of the problem. However, this class of data has an interesting property that sorting the items according to decreasing profit-to-weight ratios corresponds to a sorting of increasing weights. This leads to the investigation of specialized algorithms for efficiently solving such a problem.

- ***Inverse strongly correlated data:*** Here, for each item the profit is a linear function with a negative constant of the weight. Like the strongly correlated data problem, this data type is also hard to solve.
- ***Subset Sum data:*** This type of data can be viewed as a special case of the strongly correlated data, where profit of each item is equal to weight. Consequently, the upper bound for Subset Sum problem is the knapsack capacity. This problem seems to be hard to solve by algorithms based on branch and bound, since the bounding test is not effective in cutting off the branches. Hence, a complete enumeration may be required to obtain an optimal solution.

In this thesis we consider some problems in the knapsack family: 0-1KP, BKP and UKP associated with some hard data types such as: strongly correlated data and subset sum data. We also consider a minimization problem the so-called Change-Making problem. Our goal is to develop specialized algorithms for each problem. The quality of each algorithm will be established by a comparative analysis. We introduce the terminology used throughout this thesis in Section 1.2. A review and outline of the thesis is presented in Section 1.3.

1.2 Terminology

For the Bounded Knapsack Problem, the Unbounded Knapsack Problem and the Change-Making Problem, we use “*item type*” instead of “*item*” as being used in the 0-1Knapsack Problem. The number of items or item types will be indicated

by n . The value associated with the i^{th} item will be called *profit* and denoted by p_i ($i = 1, 2, \dots, n$). For the Change-Making problem, the i^{th} denomination of coins will also be called the *weight* as in the general Knapsack Problem and is denoted by w_i ($i = 1, 2, \dots, n$). As in the literature, it is assumed without loss of generality that profits, weights and knapsack capacity are positive integers.

To measure the difficulty in solving a problem, there are three classes of problems: P , NP and NP -Complete. Problems for which polynomial time algorithms are known are classified as belonging to the class P . This means that there exist algorithms that can solve all instances of the problems using a maximum number of steps that increases polynomially with some measure of the problem size. The class NP consists of all the problems in P as well as other problems that can be solved by a non-deterministic algorithm in polynomial time. The NP -Complete class is a subset of NP having the property that all problems in NP can be reduced, in polynomial time, to one of them. We say that a problem Q can be reduced in polynomial time to problem R if any instance of Q can be transformed in polynomial time in to an instance of R , such that the solution of Q can be obtained in polynomial time from the solution of the instance of R . A problem is ***NP-hard*** if every problem in NP is polynomially reducible to it. All Knapsack Problems considered in this thesis are *NP-hard*.

A heuristic is an algorithm that is not guaranteed to find optimal solutions. For *NP-complete* problems where computationally efficient exact algorithms are unlikely to exist, heuristic algorithms are often produced. For a maximization problem, denote an optimal solution value as z^* and the value returned by a heuristic algorithm H as z^H . If $z^H \geq (1-\varepsilon)z^*$ where $0 \leq \varepsilon \leq 1$, then the algorithm H is called a $(1-\varepsilon)$ - *approximation algorithm*. A heuristic algorithm, which returns a solution to a problem in a polynomial time, is called a *polynomial time approximation algorithm*. If the running time is also polynomial in $1/\varepsilon$, the algorithm is then called a *fully polynomial time approximation scheme*.

In the detail descriptions of all algorithms presented in this thesis, the statement “*go to*” (which should be, of course, avoided in the C language code) is only used to make understanding of the steps or iterations in the procedure descriptions clearer.

1.3 Review and Outline of the Thesis

Basically, all algorithms for Knapsack Problems require sorting the profit-to-weight ratios of items. In fact, much of the computational time required for solving Knapsack Problems is attributed to sorting. Consequently, in the last two decades, considerable effort has gone into developing algorithms which sort only a chosen small subset of items the so-called “*core*”. This has led to the solution of large-sized Knapsack Problems.

This new approach was first presented by Balas & Zemel (1980). The authors developed a method to determine an exact solution to the Knapsack Problem by solving the “*core problem*”. Further, two effective algorithms for solving 0-1KP without complete sorting have been derived from the *Balas-Zemel idea* by Fayard & Plateau (1982) and Martello & Toth (1988). Most exact algorithms for many knapsack-type problems utilize the technique of solving a core problem to speed up the solving procedure. For instance, using the branch and bound approach, Martello & Toth (1990a) developed specialized algorithms for the 0-1KP (algorithm MT2), the BKP (algorithm MTB2), the UKP (algorithm MTU2) and the CMP (algorithm MTC2). Pisinger (1995b) developed algorithms, based on dynamic programming, for the 0-1KP (algorithm MINKNAP), the BKP (algorithm BOUKNAP). Recently, an effective exact algorithm, called COMBO, for the 0-1KP has been developed by Martello, Pisinger & Toth (1997). This algorithm combines the core problem with a maximum cardinality constraint (in terms of the number of items in an optimal solution).

The approximation methods for the classical Knapsack Problems have been developed since Sahni (1975) obtained the first polynomial-time approximation scheme for 0-1KP. In the same year, the first fully polynomial-time approximation scheme was presented by Ibarra & Kim (1975). Recently, Capara *et al.* (1998) presented some improvements of this method. The authors proposed a number of linear-storage polynomial time approximation algorithms and a dynamic programming based fully polynomial time for some types of Knapsack Problems.

The computational results of several algorithms for Knapsack Problems indicate that the higher the correlation data type the more difficult the problems. The strongly correlated and the subset sum data instances of Knapsack Problems seem to be hard problems. Chvatal (1980) proved that if all coefficients of a 0-1KP are exponentially growing, and if the profit equals the weight (subset sum data), then no bounding and no dominance relations will stop the enumeration process before at least $2^{n/10}$ subproblems have been solved. This has motivated some researchers (for example Chung, Hung & Rom (1988); Chvatal(1980); Pandit & Ravi Kumar (1993); and recently Pisinger (1998)) towards concentrating only on the solution of hard Knapsack Problems. Bjorndal *et al.* (1995) suggested that future research should be concentrated on the solution of complex Knapsack Problems as well as hard data instances since results obtained for easy Knapsack Problems have been so good, and widely known.

One of the special instances of simple Knapsack Problem is the Change-Making Problem. Firstly, the problem (as defined in Section 1.1) was considered as a special case of the general, integer linear programming problem by Gomory in 1958 (Chang & Gill 1970). The Change-Making Problem has been studied since 1970 from both a theoretical and a practical point of view. However, very few exact algorithms have been proposed for solving this problem. Most recently, Martello & Toth (1990a) developed their algorithms MTC2 and MTCB for large-sized Change-Making problems by utilizing the idea of Balas & Zemel (1980).

The new approach of solving the core problem proposed by Balas and Zemel (1980) is based on the property that only a few variables (items) in the optimal solution of a Knapsack Problem have the values different from the solution of the continuous relaxed problem. We also utilize this property to develop a number of specialized algorithms for solving some hard Knapsack Problems.

This thesis focuses on the classical Knapsack Problems which are hard to solve efficiently by a general exact algorithm. The Change-Making Problem is also considered since there are currently few available efficient algorithms for it.

The objective of this thesis is to:

- develop specialized algorithms for the hard instances of large-sized classical Knapsack Problems: 0-1 Knapsack Problem, Bounded Knapsack Problem and Unbounded Knapsack Problem.
- develop exact algorithm for the large-sized Change-Making problem.

A literature survey consisting of relevant work that has been carried out to date is given in **Chapter 2**. This chapter devotes a section to applications of Knapsack Problems. Here applications include the role of the 0-1KP, the BKP, and the UKP in the solution of Integer Linear programming Problem (ILP). Several sections are devoted to the solution of the 0-1KP. This discussion includes: significant properties; upper bound improvements; reduction procedures; and exact and approximation algorithms.

The remaining chapters detail a number of specialized algorithms for solving some hard Knapsack Problems. Our algorithms are implemented in ANSI C programming language on a Silicon Graphic Workstation (SGI R 5000) running at clock speed of 150 MHz. All algorithms are tested on randomly generated data and a comparative analysis is carried out with available literature codes.

Chapter 3 proposes algorithms for the hard 0-1KP's instances of subset sum, strongly correlated, and inverse strongly correlated. The algorithms for the hard Bounded Knapsack Problem instances of strongly correlated and subset sum are also presented. Actually, these problems can be exactly solved by the recent effective algorithms for general data type such as MT2 (Martello & Toth, 1988), MINKNAP (Pisinger, 1995b) and COMBO (Martello, Pisinger & Toth, 1997) each of which utilize the concept of solving the core problem. Among these algorithms COMBO is the most efficient algorithm for the 0-1KP as demonstrated in Martello, Pisinger & Toth (1997). For solving the Strongly Correlated Knapsack Problems, a specialized algorithm has been proposed by Pandit and Ravi Kumar (1993) based on lexicographical search. Most recently, Pisinger (1998) proposed a fast algorithm, SCKNAP, based on the idea of deriving an additional constraint from an extended cover and applying surrogate relaxation with optimal multipliers. The relaxed problem is solved to determine the optimal solution to the original problem. The computational results demonstrates that SCKNAP can solve problems with size n up to 100000 in less than 50 milliseconds when the tests were performed on a HP9000/735.

All our algorithms consist of three stages: the first stage generates an initial solution by a greedy algorithm; the second stage refines the approximate solution; and the final stage applies a partial lexicographic search procedure to generate an optimal solution. Extensive computational results show that our algorithms are able to solve large problems of size up to one million variables in less than 7 seconds. A comparative analysis with some recent effective algorithms is given.

Chapter 4 proposes algorithms for the hard Unbounded Knapsack Problems instances of strongly correlated and subset sum. The Unbounded Knapsack Problem is a generalization of the well-known 0-1 Knapsack Problem. Two algorithms one for the Unbounded Strongly Correlated Knapsack Problem (algorithm CKU1) and one for the Unbounded Subset Sum Problem (algorithm CKU2) are presented. These problems seem to be difficult to solve by using dynamic programming or branch and bound algorithms which contain a reduction

procedure as a pre-processing stage. All reduction algorithms proposed to date, are not efficient for these problems due to a large number of undominated item types. Our algorithms do not contain any reduction procedures but are based on the ideas of exchanging and searching within a small set of undominated item types. Extensive computational results show that our two algorithms are able to solve large problems with up to one million variables in less than 0.3 second. Comparative analysis with the algorithm MTU2, the well-known algorithm for UKP, is also given.

Finally, **Chapter 5** proposes exact algorithms for the Change-Making Problem. The problem is a particular type of single Knapsack Problem. We consider the Unbounded Change-Making Problem (UCMP) and the Bounded Change-Making Problem (BCMP). To date, the most efficient exact algorithms (MTC2 for the UCMP and MTCB for the BCMP) are due to Martello & Toth (1990a, p.150). The authors implemented their algorithms on an HP 9000/840. The test data used randomly generated integer weights from a uniform distribution in the interval $[1, R]$. The computational results show that MTC2 can solve the UCMP's with size n up to 20000 within 0.4 second when $R = 4n$, and within 6 seconds when $R = 10^5$. For the BCMP, MTCB can solve the problems with size n up to 10000 when $R = 4n$ within 2.5 seconds. In this chapter we propose two exact algorithms: algorithm CKUC for the UCMP and algorithm CKBC for the BCMP. A combinatorial algorithm NEXCOM for generating all compositions of a positive integer m into k parts is utilized in the searching strategy to obtain an optimal solution. The algorithms can solve large-sized problems with up to $n = 300000$ when the item types are generated in range $R = 10^3$ and $R = 10^4$ within less than 51 milliseconds for UCMP and less than 3.5 seconds for BCMP when $n = 30000$. Comparative analysis for some variants of problems is also given.

CHAPTER 2

Literature Review

There are many different types of Knapsack Problems in the literature. This thesis considers some of these problems. Typically, given a set of n item types with each item type i having a profit p_i and a weight w_i , the Knapsack Problem is to select an optimal subset of item types to fill the knapsack while satisfying all knapsack constraints. Most Knapsack Problems are maximization problems which determine the largest possible profit (value) of the knapsack. It is customary to assume that profits, weights and the capacities are positive integers. The single Knapsack Problem or one-dimensional Knapsack Problem is that of choosing some of the n item types to satisfy only one capacity constraint and such that the corresponding profit sum is maximized. This chapter gives a review of the applications and some of the algorithmic developments of the single Knapsack Problems.

2.1 Applications

Interest in Knapsack Problems arise because of a very wide range of practical applications as well as theoretical considerations. In practical applications, Knapsack Problems are used to model many real-world problems including: cargo loading, cutting stock, project selection, budget control, capital investment, and resource allocation. Some of these applications have already been mentioned in the previous chapter. On the theoretical side, several algorithms for large integer programming problems require the solution of a Knapsack Problem as a subproblem. In the following we will illustrate some of the theoretical applications on how Knapsack Problems arise in the techniques for solving more complex integer programming problems.

2.1.1 Reduction of integer linear programming to Knapsack Problem (KP)

A KP can be considered as a linear programming problem with integer variables; that is, an integer linear programming problem (ILP). There is a close relationship between the Knapsack Problem and general ILP's. A number of researchers such as Balas (1965), and Elmaghraby & Wig (1970) have established that ILP's can be transformed to an equivalent KP by aggregating constraints. Every system of linear equations with integer coefficients can be transformed to a single linear equation that has the same set of nonnegative integer solutions as the original system. This implies that the constraints of an integer program can first be transformed to a single knapsack constraint. Thus any algorithm for solving the KP can be exploited to solve the transformed problem. The transformation may be worthwhile in solving the ILP since the equivalent transformed KP can be solved in pseudo-polynomial time by a number of algorithms. To utilize the reduction of an ILP to a KP, the aggregating process has been improved by Glover (1972) and

Kendell & Zionts (1972). One of the transformation processes can be illustrated as follows (see Syslo, Deo & Kowalik 1983, pp121-122):

Assume that variables of an ILP are bounded and suppose there are two equality constraints as:

$$\sum_{i=1}^n w_{1i} x_i = c_1, \quad (2.1)$$

and

$$\sum_{i=1}^n w_{2i} x_i = c_2, \quad (2.2)$$

where $0 \leq x_i \leq b_i$, x_i an integer, $i = 1, 2, \dots, n$.

$$\text{Let } g_1(x) = \sum_{i=1}^n w_{1i} x_i - c_1 = 0 \quad \text{and} \quad g_2(x) = \sum_{i=1}^n w_{2i} x_i - c_2 = 0. \quad (2.3)$$

The transformation method aggregates the two equations by choosing a nonzero integer λ , so-called valid multiplier, such that

$$g_1(x) + \lambda g_2(x) = 0 \quad (2.4)$$

implies

$$g_1(x) = g_2(x) = 0. \quad (2.5)$$

Consider the bounds on $g_1(x)$ and $g_2(x)$ that can be derived from the upper bound of x_i as follows:

$$c_j - \sum_{i=1}^n w_{ji}^+ b_i < g_j(x) < c_j - \sum_{i=1}^n w_{ji}^- b_i, \quad j = 1, 2 \quad (2.6)$$

where $w_{ji}^+ = \max(w_{ji}, 0)$ and $w_{ji}^- = \min(w_{ji}, 0)$.

This means that if we choose λ satisfying

$$\lambda > \max \left\{ c_1 - \sum_{i=1}^n w_{1i}^- b_i, -c_1 + \sum_{i=1}^n w_{1i}^+ b_i \right\}, \quad (2.7)$$

then the equation (2.4) has the same solution vector as the system of equations (2.3). If we apply the transformation to an ILP with two equality constraints (P1):

$$\begin{aligned}
(P1): \quad & \text{maximize} && \sum_{i=1}^n p_i x_i \\
& \text{subject to} && \sum_{i=1}^n w_{1i} x_i = c_1, \\
& && \sum_{i=1}^n w_{2i} x_i = c_2, \\
& && 0 \leq x_i \leq b_i, \quad x_i \text{ an integer, } i = 1, 2, \dots, n,
\end{aligned}$$

then the equivalent BKP is obtained as:

$$\begin{aligned}
(P2): \quad & \text{maximize} && \sum_{i=1}^n p_i x_i \\
& \text{subject to} && \sum_{i=1}^n (w_{1i} + \lambda w_{2i}) x_i = c_1 + \lambda c_2, \\
& && 0 \leq x_i \leq b_i, \quad x_i \text{ an integer, } i = 1, 2, \dots, n,
\end{aligned}$$

where λ is chosen satisfying (2.7).

Hence, we can solve P2 instead of P1 by any available algorithm for KP. An arbitrary system of equations can be repeatedly aggregated by considering pairs of equations and transforming them into a single equation. However this transformation approach is of limited use due to the rapid growth in the values of the coefficients.

2.1.2 Upper bound for some integer programming problems

The 0-1KP has a very wide range of applications in solving not only other types of discrete optimization problems, but also some types of problems in the knapsack family. We discuss some of these below.

Surrogate relaxation of 0-1 Multiple Knapsack Problem (MKP)

The MKP can be formulated as:

$$\begin{aligned}
 & \text{maximize} && \sum_{i=1}^m \sum_{j=1}^n p_{ij} x_{ij} \\
 & \text{subject to} && \sum_{j=1}^n w_j x_{ij} \leq c_i, && i = 1, \dots, m, \\
 & && \sum_{i=1}^m x_{ij} \leq 1, && j = 1, \dots, n, \\
 & && x_{ij} = 0 \text{ or } 1, && \text{for all } i \text{ and } j,
 \end{aligned}$$

where $x_{ij} = \begin{cases} 1 & , \text{if item } j \text{ is chosen for knapsack } i \\ 0 & , \text{otherwise.} \end{cases}$

Clearly, when $m=1$, MKP reduces to 0-1KP. Given a positive vector $(\pi_1, \pi_2, \dots, \pi_m)$ of multipliers, the standard Surrogate relaxation of MKP, S(MKP), is:

$$\begin{aligned}
 \text{S(MKP):} \quad & \text{maximize} && \sum_{i=1}^m \sum_{j=1}^n p_j x_{ij} \\
 & \text{subject to} && \sum_{i=1}^m \pi_i \sum_{j=1}^n w_j x_{ij} \leq \sum_{i=1}^m \pi_i c_i \\
 & && \sum_{i=1}^m x_{ij} \leq 1, && j = 1, 2, \dots, n, \\
 & && x_{ij} = 0 \text{ or } 1, && \text{for all } i \text{ and } j.
 \end{aligned}$$

Martello & Toth (1990a, pp.158-159) have proved that the optimal multipliers for the S(MKP) are $\pi_i = k$ ($k > 0$), $i = 1, 2, \dots, m$. Hence S(MKP) is equivalent to 0-1KP. By setting $y_j = \sum_{i=1}^m x_{ij}$ and $c = \sum_{i=1}^m c_i$, $j = 1, 2, \dots, n$, the S(MKP) can be rewritten as:

$$\begin{aligned}
& \text{maximize} && \sum_{j=1}^n p_j y_j \\
& \text{subject to} && \sum_{j=1}^n w_j y_j \leq c, \\
& && y_j = 0 \text{ or } 1, \quad j = 1, 2, \dots, n.
\end{aligned}$$

The authors have also demonstrated that the upper bound of the continuous relaxation of S(MKP) is equal to that of the continuous relaxation of MKP. However, the upper bound for MKP can be easily determined by solving a continuous 0-1KP.

Lagrangian relaxation of 0-1 Multiple Knapsack Problem

Suppose a vector $(\lambda_1, \lambda_2, \dots, \lambda_n)$ of nonnegative multipliers is given, the Lagrangian relaxation of MKP, $L(\text{MKP}, \lambda)$, is

$$\begin{aligned}
L(\text{MKP}, \lambda): \quad & \text{maximize} && \sum_{i=1}^m \sum_{j=1}^n p_j x_{ij} - \sum_{j=1}^n \lambda_j \left(\sum_{i=1}^m x_{ij} - 1 \right) \\
& \text{subject to} && \sum_{j=1}^n w_j x_{ij} \leq c_i, \quad i = 1, 2, \dots, m, \\
& && x_{ij} = 0 \text{ or } 1, \quad i = 1, 2, \dots, m \text{ and } j = 1, 2, \dots, n.
\end{aligned}$$

If we set $\hat{p}_j = p_j - \lambda_j$, the problem $L(\text{MKP}, \lambda)$ can be rewritten as:

$$\begin{aligned}
& \text{maximize} && \sum_{i=1}^m \sum_{j=1}^n \hat{p}_j x_{ij} + \sum_{j=1}^n \lambda_j \\
& \text{subject to} && \sum_{j=1}^n w_j x_{ij} \leq c_i, \quad i = 1, 2, \dots, m, \\
& && x_{ij} = 0 \text{ or } 1, \quad i = 1, 2, \dots, m \text{ and } j = 1, 2, \dots, n.
\end{aligned}$$

Therefore, if we solve m 0-1KP's:

$$\begin{aligned}
&\text{maximize} && z_i = \sum_{j=1}^n \hat{p}_j x_{ij} \\
&\text{subject to} && \sum_{j=1}^n w_j x_{ij} \leq c_i, \\
&&& x_{ij} = 0 \text{ or } 1, \quad j = 1, 2, \dots, n,
\end{aligned}$$

then we will obtain the solution of the L(MKP, λ) as:

$$z(L, \lambda) = \sum_{i=1}^m z_i + \sum_{j=1}^n \lambda_j.$$

Actually, it is not known how to determine analytically the vector (λ_j) producing the lowest possible value of $z(L(MKP, \lambda))$. However, we may tentatively choose λ_j as

$$\bar{\lambda}_j = \begin{cases} p_j - w_j \frac{p_s}{w_s} & , \text{ if } j < s \\ 0 & , \text{ otherwise} \end{cases}$$

where $s = \min \left\{ j : \sum_{k=1}^j w_k > c \right\}$, $c = \sum_{i=1}^m c_i$, then the continuous solution of the L(MKP, $\bar{\lambda}$) is equal to the continuous solution of S(MKP) and dominates the continuous relaxation of MKP.

Upper bound for the Multiconstraint 0-1 Knapsack Problem

Many researchers such as Shih (1979) and Gavish & Pirkul (1985) utilize the 0-1KP in estimating an upper bound for the Multiconstraint 0-1 Knapsack Problem. The problem may be formulated as:

$$\begin{aligned}
&\text{maximize} && \sum_{j=1}^n p_j x_j \\
&\text{subject to} && \sum_{j=1}^n w_{ij} x_j \leq c_i, \\
&&& x_j = 0 \text{ or } 1, \quad j = 1, 2, \dots, n.
\end{aligned}$$

Shih (1979) simply treats the problem as m 0-1KP's with the same objective function. The i^{th} problem is:

$$\begin{aligned} &\text{maximize} && \sum_{j=1}^n p_j x_j \\ &\text{subject to} && \sum_{j=1}^n w_j x_j \leq c_i, \\ &&& x_j = 0 \text{ or } 1, j = 1, 2, \dots, n. \end{aligned}$$

An upper bound (*Dantzig bound*) is computed for each 0-1KP separately. The minimum objective function value obtained among those problems is used as the upper bound at each node in branch and bound algorithms. Gravish & Pirkul (1985) determine a single constraint continuous KP by a surrogate constraint and then solve the problem to generate the bound for the Multiconstraint 0-1 Knapsack Problem.

2.1.3 Cutting Plane Generation

One of the principal techniques for solving 0-1 integer program is the cutting plane method. The main idea is to solve a relaxed linear programming problem. If the resulting solution satisfies the integer restriction, then it is optimal. If not, then a constraint (cut) is added that eliminates the candidate solution from further consideration. Several methods for generating a set of valid inequalities or the so-called *cuts* have been investigated over the past two decades. Many authors have proposed a number of procedures to generate and strengthen the cuts.

Suppose $\sum_{j=1}^n a_j x_j \leq L$ is the original inequality, where L is a positive integer.

Kianfar (1971) uses knapsack functions to strengthen a cut by defining the following knapsack function $F_n^r(k)$ for all integer k , $0 \leq k \leq L$.

$$F_n^r(k) = \max \sum_{j=1, j \neq r}^n a_j x_j$$

$$\text{subject to } \sum_{j=1, j \neq r}^n a_j x_j \leq k, \quad 0 \leq k \leq L,$$

$$x_j \in \{0, 1, 2, \dots\}, \quad j = 1, 2, \dots, n,$$

where the a_j 's are positive integers for all j .

Let $S_n^r(k) = k - F_n^r(k) \geq 0$ for $0 \leq k \leq L$. It is established that the stronger inequality is

$$\sum_{j=1, j \neq r}^n a_j x_j + (a_r + k_r)x_r \leq L, \quad x_j \in \{0, 1, 2, \dots\}, \quad j = 1, 2, \dots, n,$$

where $0 \leq k_r \leq \min_h (S_n^r(L - ha_r)/h)$, $h = 1, 2, \dots, [L/a_r]$. The author suggested

that the best value of k_r is $\min_h (S_n^r(L - ha_r)/h)$, $h = 1, 2, \dots, [L/a_r]$.

Balas (1975) gave a necessary and sufficient condition for an inequality with coefficients 0 and 1 to define a facet of the knapsack polytope, i.e the convex hull of 0-1 points satisfying the knapsack constraint:

$$P = \text{conv}\{x \in \mathcal{R}^n \mid \sum_{j=1}^n a_j x_j \leq b; x_j = 0 \text{ or } 1, j = 1, 2, \dots, n\}. \quad (2.8)$$

The author also proposed a procedure for generating all facets that can be generalized to general cutting planes for 0-1 programs.

Crowder, Johnson & Padberg (1983) proposed a branch and cut procedure to solve large-scale zero-one linear programming problems (IP): minimize $\{cx \mid Ax \leq b; x_j = 0 \text{ or } 1 \text{ for } j = 1, 2, \dots, n\}$; where A is an $m \times n$ matrix with arbitrary rational entries, b and c are $m \times 1$ and $1 \times n$ matrices respectively. The procedure makes use of a series of knapsack inequalities in order to chop off fractional solutions during the running of the cutting plane algorithm. Let P_1^i be the knapsack polytope

associated with constraint i , $i = 1, 2, \dots, m$ of the IP and the zero-one polytope $P_I = \text{conv}\{x \in \mathcal{R}^n \mid Ax \leq b; x_j = 0 \text{ or } 1 \text{ for } j = 1, 2, \dots, n\}$. Clearly, we can see that $P_I \subseteq \bigcap_{i=1}^m P_I^i$. The authors remarked that “if we have a large-scale zero-one programming problem with a sparse matrix A and with no apparent special structure, then the intersection of the m knapsack polytopes P_I^i provides a fairly good approximation to P_I .” They also presented a recursive lifting procedure, which preserves the sparsity of the constraint matrix. The procedure requires several relaxation solutions of 0-1KP’s. Moreover, the authors established that the linear program can be strengthened by adding violating knapsack inequalities.

Van Roy & Wolsey (1987) presented a relatively new approach: the use of strong cutting planes for reformulating and solving mixed 0-1 programs which are viewed as Fixed Charge Network structures. The solution of 0-1KP is required in making a good choice of a cover for the process of generating cover inequalities and extending generalized flow cover inequalities.

2.1.4 Penalty estimation of Integer programming

In integer programming, the original idea of the so-called “*penalty*” is to underestimate the amount by which the optimal value of the lower bound relaxation of the current candidate problem in a branch and bound approach would increase. The estimates of change, referred to as *penalties*, can be used as a criterion to fathom the unpromising candidate problem in a reduction procedure. Moreover the strengthened penalty may be helpful to generate some types of cutting-plane.

Geoffrion (1974) derived the standard penalties of Driebeek (1966) and Tomlin (1970, 1971) and also developed several new penalties from the viewpoint of Lagrangean relaxation. The computation of bounds for the optimal value of candidate problem with conditional bound of each fractional variable in the LP

solution of a candidate problem requires solving a set of KP's with some integer variables; see the Geoffrion (1974) for detail.

Although KP's are *NP-hard*, efficient algorithms for solving several types of problems have been developed. Since the 0-1KP is the most important, not only in the family of Knapsack Problems but also in discrete optimization problems, it is worthwhile to study some significant properties of the 0-1KP and the development of the results for this problem.

2.2 Connection with Networks

In 1967, Shapiro & Wagner (1967) were the first who studied the mathematical and algorithmic connections between an UKP with an equality constraint and a fundamental renewal-type problem that can be solved by a network approach. The authors showed that the UKP could be solved by means of solving the renewal problem. The UKP is describe as:

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^n p_i x_i \\ & \text{subject to} && \sum_{i=1}^n t_i x_i = T, \\ & && x_i \text{ nonnegative integer, } i = 1, 2, \dots, n, \end{aligned}$$

where p_i is a finite real number, T and t_i , $i = 1, 2, \dots, n$ are positive integers.

The renewal problem can be illustrated as an economic problem in the theory of growth. Consider an economy with n production policies. The profit p_i is yielded by policy i and extends over t_i time periods. The optimization problem is to choose a time sequence of policies extending over the planning horizon T that maximizes the present value of profit returns. A constant α , $0 \leq \alpha \leq 1$ is the usual discount factor. By assuming that p_i is available at the initial period, p_i may represent a present value of profit return over the t_i periods. $G(T)$ is the present

value of an optimal sequence of policies called an optimal strategy when the horizon length is T periods; $T = 1, 2, 3, \dots$ and $G(0) = 0$.

The renewal-type problem is then formulated in terms of dynamic programming recursions as:

$$G(T) = \max_i \left\{ p_i + \alpha^{t_i} G(T - t_i) \right\}, \quad i = 1, 2, \dots, n,$$

where $t_i \leq T$, $0 \leq \alpha \leq 1$ and $G(0) = 0$. The solution of the renewal problem can be expressed in network terms as:

Let the nodes of the network be denoted as $0, 1, 2, \dots, T$,

$(T - t_i, T)$: the directed arc i , where $t_i < T$ that represents a feasible policy decision at the initial period T ,

$G(T)$: the value of an optimal path from the origin to node T ,

$p_i + \alpha^{t_i} G(T - t_i)$: the present value of deciding on policy i in a T -period horizon and acting optimally thereafter.

Suppose an acyclic network is a network such that each arc $(K - t_i, K)$ where $t_i \leq K$, corresponds to a possible decision when there are K periods remaining in the horizon. The Network optimization problem is to determine an optimal tree of best directed paths from the origin ($T = 0$) to node T ($T = 1, 2, 3, \dots$) of an acyclic network. Then the UKP can be solved by taking $\alpha = 1$. Shapiro & Wagner (1967) proposed an efficient algorithm called *Renewal algorithm* to solve the renewal problem by exploiting a network approach and they applied this method immediately to the UKP. The details of the algorithm can be seen in Shapiro & Wagner (1967).

Shapiro (1968) has developed an efficient algorithm based on the renewal and knapsack algorithm, for solving the group optimization problem. Moreover, the method can be extended to an algorithm for solving all integer programming problems. The author utilized a network approach to construct the algorithm in the

sense of a shortest route problem and this approach is illustrated by Salkin (1975, pp. 422-423).

2.3 Some Properties of the 0-1Knapsack Problem

An important property of the 0-1KP is that an upper bound can be obtained from an optimal solution to the continuous relaxed problem. Consider the 0-1KP by assuming that all values p_i and w_i are positive integers, $w_i < c$, $i = 1, 2, \dots, n$ and $\sum_{i=1}^n w_i > c$ where c is a given capacity of the knapsack. An arbitrary 0-1KP problem can be transformed to the equivalent one satisfying these assumptions. Dantzig (1957) established a simple way of determining an optimal solution to the *continuous* KP, C(KP), with the constraint $x_i = 0,1$ relaxed to $0 \leq x_i \leq 1$, $i=1,2,\dots, n$. More specifically, he considered the problem:

$$\begin{aligned} \text{C(KP):} \quad & \text{maximize} \quad \sum_{i=1}^n p_i x_i \\ & \text{subject to} \quad \sum_{i=1}^n w_i x_i \leq c, \\ & \quad \quad \quad 0 \leq x_i \leq 1, \quad i = 1, 2, \dots, n. \end{aligned}$$

By using a greedy algorithm, the successive items with largest profit-to-weight ratio are chosen to fill the knapsack until the next item b which does not fit into the knapsack is reached. The item b is called the *critical variable* or *break item*. Assuming that the items (variables) are ordered in terms of profit- to-weight ratio as: $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$, the optimal solution to the C(KP) is:

$$\left. \begin{aligned} x_i^* &= 1, & , i &= 1, 2, \dots, b-1, \\ x_b^* &= \left(c - \sum_{i=1}^{b-1} w_i \right) / w_b, \\ x_i^* &= 0, & , i &= b+1, b+2, \dots, n, \end{aligned} \right\} \quad (2.9)$$

where $b = \min \left\{ j \mid \sum_{i=1}^j w_i > c \right\}$. The optimal solution value of C(KP) is:

$$z(C(KP)) = \sum_{i=1}^{b-1} p_i + \left(c - \sum_{i=1}^{b-1} w_i \right) \frac{p_b}{w_b}.$$

This optimal solution immediately gives rise to the following upper bound for the

0-1KP as: $U_1 = \sum_{i=1}^{b-1} p_i + \left\lfloor \left(c - \sum_{i=1}^{b-1} w_i \right) \frac{p_b}{w_b} \right\rfloor$. This bound is the so-called *Dantzig*

bound. Several modified upper bounds have been proposed by Balas & Zemel (1980) and Martello & Toth (1977, 1988). There are discussed in the next section.

Another important property is that only a few variables in the solution to the continuous relaxed problem differ from the optimal integer solution. Moreover the different variables are around the break item, thus to obtain an optimal integer solution only a few variables need to be changed. This property leads to the development of many algorithms for solving Knapsack Problems. Balas & Zemel (1980) were the first to propose a “*core problem*” to determine an optimal solution to large-sized 0-1KP by a dynamic programming procedure. The core is a small subset of all items and can be constructed, using a heuristic algorithm, without sorting all profit-to-weight ratios. The most efficient algorithms for several types of large-sized Knapsack Problems developed by Martello & Toth (1990a) obtain an optimal solution in pseudo-polynomial time. These algorithms solve the core problem through a dynamic programming procedure. Moreover, solving the C(KP) with an additional maximum cardinality constraint:

$$\sum_{i=1}^n x_i \leq b' - 1 \quad (2.10)$$

where $b' = \min_{j \in N} \left\{ j \mid \sum_{i=1}^j w_i > c; w_1 \leq w_2 \leq \dots \leq w_n \right\}$, yields an upper bound stronger

than U_1 . In addition the optimal solution to the C(KP) with the additional constraint (2.10), C(KPA), contains two fractional variables and the number of variables whose values differ from the optimal solution to C(KPA) is less than that of C(KP). The following table illustrates an optimal solution to a 0-1KP compared to the continuous solutions of C(KP) and C(KPA).

variables (x_i)	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	objective value
$c = 294, n = 12$													
p_i	35	79	39	72	70	28	62	66	56	56	47	41	
w_i	3	12	9	20	33	26	58	66	73	96	98	94	
continuous sol. of C(KP)	1	1	1	1	1	1	1	1	0.918	0	0	0	502.397
continuous sol. of C(KPA)	0.785	1	1	1	1	0	1	1	1	0.215	0	0	483.516
optimal sol. of 0-1KP	1	1	1	1	1	0	1	1	1	0	0	0	479

Table 2.2 : An optimal solution to the 0-1 KP compares to the continuous solution of C(KP) and C(KPA). All problems were solved by the CPLEX (LP solver).

Table 2.2 shows that the break item is $b' = 8$, and the objective value corresponding to the optimal solution to the C(KPA) is very close to that of 0-1KP. Only two variables in both continuous problems have their values different from the integer solution of the 0-1KP. Especially, since there are two fractional variables x_1 and x_{10} in the solution to the C(KPA), we may guess that x_1 is more likely to be in the knapsack than x_{10} .

This property leads to the fully polynomial time approximation scheme of Capara *et al.*(1998). The authors propose a heuristic algorithm $H^{1/2}$ for the 0-1KP with the additional constraint (2.10), KPA, that is based on the optimal solution of the C(KPA). They also give a proof of the following lemma:

Lemma 2.5.1 An optimal basic solution (x^*) of the C(KPA) has at most two fractional components. Let $J_I = \{ k : x_k^* = 1 \}$, $p(J_I) = \sum_{k \in J_I} p_k$ and z^* be the optimal

value. If the basic solution has two components x_i^* and x_j^* , supposing without loss of generality $w_j \leq w_i$, then $p(J_I) + p_i \geq z^*$ and the solution defined by $J_I \cup \{j\}$ is feasible for the KPA.

The algorithm $H^{1/2}$ is proved as a $1/2$ approximation algorithm (see the definition in Section 1.2) for the KPA and runs in $O(n)$ time since $z^H/z^* \geq 1/2$ where z^H is the value of the solution returned by the heuristic algorithm $H^{1/2}$. The authors also modified a polynomial time approximation scheme for 0-1KP, Subset-sum problem and m -dimensional knapsack problem by utilizing the procedure $H^{1/2}$.

Another interesting property of the 0-1KP is that it is separable. Horowitz & Sahni (1974) established that the 0-1KP can be solved to optimality by dividing the problem into 2 subproblems or, in other words, the set of all variables is divided into 2 subsets. Their algorithm enumerates all feasible solutions in each subset and the optimal solution is obtained by merging the two sets of feasible solutions and determining the best one. The 0-1KP can be solved in $O(\sqrt{2^n})$. Moreover, the solution method can be improved over a complete enumeration by a factor of a square root. This leads to a development of parallel algorithms for solving the 0-1KP by recursively dividing the problem to the smaller size.

2.4 Upper Bound Improvements

When solving any combinatorial optimization problem by branch and bound or dynamic programming procedure, a good upper bound is required. The 0-1KP is a combinatorial optimization problem that has a lot of upper bound improvements. As mentioned above, the simple *Dantzig bound* has given rise to numerous upper bound modifications. These are discussed in the following.

2.4.1 Bounds from Additional Constraints of some variables

Martello & Toth (1997) improved the *Dantzig bound* by imposing the integrality of the break item x_b . In the optimal solution to the C(KP), the break item has fractional value between 0 and 1; thus in an optimal solution to the 0-1KP either $x_b = 0$ or $x_b = 1$. Firstly assuming $x_b = 0$ the next item $b+1$ can be inserted by relaxing the integral constraint of x_{b+1} to $x_{b+1} > 0$ yielding an upper bound on 0-1KP as:

$$U^0 = \bar{p} + \left\lfloor (c - \bar{w}) \frac{p_{b+1}}{w_{b+1}} \right\rfloor$$

where $\bar{p} = \sum_{i=1}^{b-1} p_i$ and $\bar{w} = \sum_{i=1}^{b-1} w_i$. When $x_b = 1$, we need to remove an item from the knapsack of the weight at least $w_b - (c - \bar{w})$ of the worst possible value $\frac{p_{b-1}}{w_{b-1}}$. Hence the bound can be

$$U^1 = \bar{p} + \left\lfloor p_b - (w_b - (c - \bar{w})) \frac{p_{b-1}}{w_{b-1}} \right\rfloor.$$

Here the *Martello and Toth upper bound* on the objective value of the 0-1KP is given by $U_2 = \max(U^0, U^1)$ and it has been proved that $U_2 \leq U_1$ (Martello & Toth 1990a, pp.20-21).

Fayard & Plateau (1982) derived the upper bound U_3 for the 0-1KP based on the binary value of the break item as:

$$U_3 = \left\lfloor \max \{ z(C(KP | x_b = 1)), z(C(KP | x_b = 0)) \} \right\rfloor$$

where $z(C(KP | x_b = \varepsilon))$, $\varepsilon = 0, 1$, is optimal value of C(KP) with additional constraint $x_b = \varepsilon$, $\varepsilon = 0, 1$. Clearly $U_3 \leq U_2$ since $z(C(KP | x_b = 0)) \leq U^0$ and $z(C(KP | x_b = 1)) \leq U^1$.

2.4.2 Bounds from Lagrangian Relaxation

The Lagrangian approach can be used to produce a relaxed problem of any optimization problem that is easier to solve. Moreover, its optimal value is an upper bound (for maximization problems) on the optimal value of the original problem. Given a nonnegative multiplier λ , the Lagrangian relaxation of 0-1KP is

$$\begin{aligned} \text{L(KP, } \lambda): \quad & \text{maximize} \quad \sum_{i=1}^n p_i x_i + \lambda \left(c - \sum_{i=1}^n w_i x_i \right) \\ & \text{subject to} \quad x_i = 0 \text{ or } 1, \quad i = 1, 2, \dots, n. \end{aligned}$$

The objective value of $\text{L(KP, } \lambda)$ may be rewritten as

$$z(\text{L(KP, } \lambda)) = \{ \lambda c + \max_{i=1}^n (p_i - \lambda w_i) x_i \mid x_i = 0 \text{ or } 1, i = 1, 2, \dots, n \}$$

Its linear programming relaxation $\text{C(L(KP, } \lambda))$ which can be stated as:

$$\begin{aligned} \text{C(L(KP, } \lambda)): \quad & \text{maximize} \quad \sum_{i=1}^n p_i x_i + \lambda \left(c - \sum_{i=1}^n w_i x_i \right) \\ & \text{subject to} \quad 0 \leq x_i \leq 1, \quad i = 1, 2, \dots, n, \end{aligned}$$

is easily solvable to optimality. The optimal solution to $\text{C(L(KP, } \lambda))$ can be determined as

- (i) if $p_i - \lambda w_i > 0$, then $x_i^* = 1$,
- (ii) if $p_i - \lambda w_i < 0$, then $x_i^* = 0$,
- (iii) if $p_i - \lambda w_i = 0$, then $0 \leq x_i^* \leq 1$,

and the corresponding objective value is

$$z(\text{C(L(KP, } \lambda))) = \sum_{\{i: p_i/w_i > \lambda\}} (p_i - \lambda w_i) + \lambda c.$$

For any $\lambda \geq 0$ we have

$$z(L(KP, \lambda)) = z(C(L(KP, \lambda))) \geq z(C(KP)) \geq z(KP)$$

and it is well-known that $\lambda = \lambda^* = p_b/w_b$ yields the minimum value of $z(L(KP, \lambda^*))$ which is equal to $z(C(KP))$. Even though the bound obtained by Lagrangian relaxation will never be stronger than the *Dantzig bound* (i.e $z(C(KP))$), the value

$$\left| p_i - w_i \frac{p_b}{w_b} \right| \text{ is a lower bound on the decrease of } z(C(KP)) \text{ corresponding to the}$$

change of the i^{th} variable from x_i^* to $1-x_i^*$. Müller-Merbach (1978) and Dudzinski & Walukiewicz (1987) derived their 0-1KP upper bounds based on this idea through the Lagrangian approach.

Müller-Merbach (1978) derived an upper bound for 0-1KP by using the optimal solution to C(KP). The *Müller-Merbach bound* is based on the idea of considering (2.9) in order to obtain an integer solution of 0-1KP by either (i) setting the value of break item b in (2.9) equal to 0 i.e. $x_b^* = 0$, or (ii) a variable x_i^* , $i = 1, 2, \dots, n$ and $i \neq b$ is to be changed to $1 - x_i^*$. In case (i) we have the value of the objective function as $\bar{p}$. In case (ii) if x_b^* is relaxed as $x_b^* > 0$, then an upper bound may be computed as:

$$(1) \ u_i^0 = \bar{p} - p_i + (c - \bar{w} + w_i)p_b/w_b, \text{ by setting } x_i^* = 0, \text{ for } i < b$$

$$(2) \ u_i^1 = \bar{p} + p_i + (c - \bar{w} - w_i)p_b/w_b, \text{ by setting } x_i^* = 1, \text{ for } i > b.$$

Thus we have

$$U_4 = \max \left\{ \bar{p}, \max_{i < b} \lfloor u_i^0 \rfloor, \max_{i > b} \lfloor u_i^1 \rfloor \right\}.$$

The time complexity for the computation of U_1, U_2, U_3 and U_4 is $O(n)$, once the break item is known. Now it is easy to see that $U_4 \leq U_1$. There is no dominance between U_3 and U_4 since there are examples of showing that $U_4 > U_2 = U_3$ (Martello & Toth 1990a, pp. 23-24) and $U_4 < U_3 \leq U_2$. Thus we may consider

$$U_5 = \min\{U_3, U_4\}.$$

Further Dudzinski & Walukiewicz (1987) have obtained an upper bound dominating all the above. By applying the procedure “*filling the knapsack*”, a feasible solution of 0-1KP can be constructed from an optimal solution of C(KP) in the following way:

Step 1 : for $j = 1, 2, \dots, b-1$; $\hat{x}_j = 1$, $N_I = \{1, 2, \dots, b-1\}$,

Step 2 : for $j = b+1, \dots, n$; if $w_j \leq c - \sum_{j \in N_I} w_j x_j$ then $\hat{x}_j = 1$, $N_I = N_I \cup \{j\}$, else

$$\hat{x}_j = 0,$$

Step 3 : $\hat{x}_b = 0$.

If $\hat{x} = (\hat{x}_1, \dots, \hat{x}_n)$ is not optimal to 0-1KP then an optimal solution can be obtained by setting the break item value as 0 or 1 and changing at least one variable $\hat{x}_j = 1 - \hat{x}_j$ for every j and $j \neq b$. Let $\tilde{p}_j = p_j - w_j \frac{p_b}{w_b}$. The Dudzinski and Walukiewicz bound can be derived as:

$$U_6 = \max \left\{ \begin{array}{l} \sum_{j=1}^n p_j \hat{x}_j, \\ \min \left(\left\lfloor z(C(KP | x_b = 1)) \right\rfloor, \max_j \left\{ \left\lfloor z(C(KP)) - \tilde{p}_j \right\rfloor : j = 1, 2, \dots, b-1 \right\} \right), \\ \min \left(\left\lfloor z(C(KP | x_b = 0)) \right\rfloor, \max_j \left\{ \left\lfloor z(C(KP)) + \tilde{p}_j \right\rfloor : j \in \hat{N} \right\} \right) \end{array} \right\}.$$

where $\hat{N} = \{j : \hat{x}_j = 0, j = 1, \dots, n, j \neq b\}$. The authors prove that U_6 is a valid upper bound for the 0-1KP and $U_6 \leq U_5$.

2.4.3 Bounds from Partial Enumeration

As one property of the 0-1KP, an optimal solution can be obtained by changing values of a few variables around the break item. This leads to a further improvement of the upper bound U_3 . Martello & Toth (1988) generalized U_3 by considering decision trees having more than two terminal nodes. A strong upper bound is derived by using partial enumeration.

Let $M \subseteq N$ be a subset of the items containing the break item and $X_M = \{x_i \in \{0,1\}, i \in M\}$. Obviously, the path of an optimal solution in the decision tree of a 0-1KP must be one of the paths through X_M and a valid upper bound is thus given by

$$U_M = \max_{\bar{x} \in X_M} (z(C(KP | \bar{x})))$$

where $z(C(KP | \bar{x}))$ is an optimal value to $C(KP)$ with additional constraint $x_i = \bar{x}_i$ for $i \in M$.

2.4.4 Bounds from Valid Inequalities

Since the early seventies, the 0-1 knapsack polytope has been investigated by many researchers, e.g. Balas (1975), Hammer, Johnson & Peled (1975), Wolsey (1975), and Balas & Zemel (1978). Recently, the generalization of 0-1 knapsack polytope has been intensively studied by Weismantel (1997). The 0-1 knapsack polytope is the convex hull of the 0-1 solution to the single inequality $\sum_{i=1}^n a_i x_i \leq b$ denoted as (2.8). One of the basic objects in polyhedral theory is *minimal covers*.

Let N be a subset of items and c denote the knapsack capacity. Suppose every item $i \in N$ has a weight $w_i > 0$, a set $C \subseteq N$ is a *cover* if $\sum_{i \in C} w_i > c$. A cover is

minimal if $\sum_{i \in S} w_i \leq c$ where $S = C - \{k\}$ for all $k \in C$. Hence, we have a valid cover inequality as (Balas 1975):

$$\sum_{i \in C} x_i < |C| - 1,$$

where $|C|$ denotes the cardinality of C . Adding a valid cover inequality to the continuous relaxation problem of an integer program will never exclude any integer solution and leads to stronger continuous bound.

By assuming that the weights are in non-decreasing order, and k is defined as

$$k = \min \left\{ j \left| \sum_{i=1}^j w_i > c \right. \right\} - 1,$$

Martello & Toth (1997) define a maximum cardinality constraint

$$\sum_{i=1}^n x_i \leq k. \quad (2.11)$$

The constraint (2.11) will be added only if $k = b - 1$ where b is the break item. Obviously, this additional constraint does not exclude any feasible solution of the 0-1KP since there are $b - 1$ items less than the break item. Moreover the objective value of $C(KP)$ with the additional constraint (2.11) is a stronger upper bound for 0-1KP than that of $C(KP)$. Since any optimal solution to $C(KP)$ which violates (2.11) will be excluded, a better bound will be obtained.

Similarly, by assuming that the profits are in non decreasing order and z is the current lower bound, a minimum cardinality constraint is defined as

$$\sum_{i=1}^n x_i \geq k \quad (2.12)$$

where $k = \max \left\{ h \left| \sum_{j=1}^h p_j \leq z \right. \right\} + 1$ for any solution with objective value larger than

z . Thus (2.12) will be added when $k = b$ to exclude the violating solutions. Solving the $C(KP)$ with additional cardinality constraints by LP solver may not be computationally efficient for large-sized problems.

Consider C(KP) with (2.11) added. Thus we have the problem (KP_k) as:

$$\begin{aligned}
 (\text{KP}_k): \quad & \text{maximize} \quad \sum_{i=1}^n p_i x_i \\
 & \text{subject to} \quad \sum_{i=1}^n w_i x_i \leq c, \\
 & \quad \quad \quad \sum_{i=1}^n x_i \leq k, \\
 & \quad \quad \quad 0 \leq x_i \leq 1, \quad i = 1, 2, \dots, n.
 \end{aligned}$$

Martello & Toth (1997) show that the KP_k can be solved using the Lagrangian approach by dualizing the additional constraints through any nonnegative Lagrangian multiplier λ . The Lagrangian problem with the relaxed integrality constraints L(KP_k, λ) is given by

$$\begin{aligned}
 L(\text{KP}_k, \lambda): \quad & \text{maximize} \quad \sum_{i=1}^n p_i x_i - \lambda \left(\sum_{i=1}^n x_i - k \right) = \lambda k + \max \sum_{i=1}^n \tilde{p}_i x_i \\
 & \text{subject to} \quad \sum_{i=1}^n w_i x_i \leq c, \\
 & \quad \quad \quad 0 \leq x_i \leq 1, \quad i = 1, 2, \dots, n,
 \end{aligned}$$

where $\tilde{p}_i = p_i - \lambda$, $i = 1, 2, \dots, n$ and $\tilde{p}_i$ may be nonpositive. Therefore, we need the optimal multiplier λ^* which yields the minimum objective value of L(KP_k, λ), i.e.

$$z(L(\text{KP}_k, \lambda^*)) = \min_{\lambda \geq 0} \{z(L(\text{KP}_k, \lambda))\}. \quad (2.13)$$

Let $C(x, \lambda)$ denote the number of items in the solution to L(KP_k, λ). Observe that for the optimal multiplier λ^*

$$|C(x, \lambda^*) - k| = \min_{\lambda \geq 0} |C(x, \lambda) - k|. \quad (2.14)$$

In order to determine the multiplier λ^* satisfying (2.14), the authors have proved that $C(x, \lambda)$ is a monotonous function of λ :

Lemma (Martello & Toth, 1997) Given two item s and t , and two values λ' and λ'' such that $\lambda' \leq \lambda''$, if

$$\frac{p_s - \lambda'}{w_s} \geq \frac{p_t - \lambda''}{w_t} \quad \text{and} \quad \frac{p_s - \lambda''}{w_s} \leq \frac{p_t - \lambda'}{w_t}$$

then $w_s \geq w_t$.

Corollary (Martello & Toth, 1997) $C(x, \lambda)$ is monotonically nonincreasing as λ is increasing.

This means that the (2.14) can be solved by binary search in determining the value of λ^* . Thus at each iteration, we define $\lambda = (\lambda_1 + \lambda_2)/2$ where λ_1 and λ_2 are a lower bound and an upper bound on λ^* respectively. By initially setting $\lambda = \lambda_1 = 0$ and $\lambda_2 = k^{\text{th}}$ largest p_i value, if $C(x, \lambda^*) = k$ then λ^* is found as $\lambda^* = \lambda$. Otherwise set $\lambda = \lambda_1$ if $C(x, \lambda) > k$ or $\lambda = \lambda_2$ if $C(x, \lambda) < k$. The procedure terminates when $\lambda_1 = \lambda_2$. Martello & Toth (1997) further modified a specialized technique based on the properties of $L(KP_k, \lambda)$ for determining the optimal multiplier λ^* in $O(n^2)$ time. Obviously,

$$U_\lambda = \lfloor z(L(KP_k, \lambda^*)) \rfloor \leq U_1$$

since $U_1 = \lfloor z(L(KP_k, \lambda = 0)) \rfloor$. The authors also derived the upper bound from the minimum cardinality in a symmetrical way as U_λ .

An alternative way to derive upper bound from cardinality constraints has been proposed by Martello, Pisinger & Toth (1997). The authors relax the cardinality constraint with the original weight constraint using surrogate multipliers $S \geq 0$ for the maximum cardinality constraint and $S \leq 0$ for the minimum cardinality constraint. In both cases, we have the relaxed problem (KP_S) as:

$$\begin{aligned}
(\text{KP}_S): \quad & \text{maximize} \quad \sum_{i=1}^n p_i x_i \\
& \text{subject to} \quad \sum_{i=1}^n (w_i + S) x_i \leq c + kS, \\
& \quad \quad \quad x_i = 0 \text{ or } 1, \quad i = 1, 2, \dots, n.
\end{aligned}$$

Clearly, the KP_S is a 0-1 KP. It also shows that for the continuous relaxed problem of KP_S , $C(\text{KP}_S)$, the optimal surrogate multiplier S^* can be derived through binary search in a similar way to the Lagrangian multiplier proposed by Martello & Toth (1997). The value $U(S^*)$ of the objective function of $C(\text{KP}_S)$ with $S = S^*$ gives the upper bound $\lfloor U(S^*) \rfloor$ which dominates the *Dantzig bound* (see Martello, Pisinger & Toth 1997).

2.5 Reduction Procedures

Most available algorithms for solving Knapsack Problems contain a reduction procedure in order to reduce the size of problems. The reduction procedures are methods of fixing as many variables as possible at their optimal values. An efficient reduction algorithm may lead to an efficient algorithm for solving the large-sized Knapsack Problem in terms of time and space complexity.

The first reduction algorithm for the 0-1KP has been proposed by Ingargiola & Korsh (1973). The *Ingargiola and Korsh reduction algorithm* (IKR) is based on the idea of the rules for branching and bounding proposed by Kolesar (1967) and Greenberg & Hegerich (1970). Assuming that, for $i, j \in N = \{1, 2, \dots, n\}$, if $i < j$ then $p_i/w_i \geq p_j/w_j$, IKR determines two disjoint subsets of N , namely $I_1 = \{j \in N \mid x_j = 1\}$ and $I_0 = \{j \in N \mid x_j = 0\}$ as follows.

Let $z = U_1 - \left(c - \sum_{j=1}^{b-1} w_j \right) \frac{p_b}{w_b}$, $j \in N$, where U_1 is the *Dantzig bound*,

$$b = \min_{j \in N} \left\{ j \mid \sum_{i=1}^j w_i > c \right\},$$

u_j^0 = an upper bound for the 0-1KP with additional constraint $x_j = 0$, and

u_j^1 = an upper bound for the 0-1KP with additional constraint $x_j = 1$.

Initially, set $I_1 = \emptyset$ and $I_0 = \emptyset$, where $\emptyset$ is the empty set.

For $j = 1, 2, \dots, n$, $I_1 = I_1 \cup \{j\}$ if $u_j^0 < z + 1$, and

$$I_0 = I_0 \cup \{j\} \text{ if } u_j^1 < z + 1.$$

Hence the remaining items are in $N \setminus (I_1 \cup I_0)$ and a solution corresponding to objective value can be found by solving the following reduced problem:

$$\begin{aligned} \text{maximize} \quad & \hat{z} = \sum_{j \in N \setminus (I_0 \cup I_1)} p_j x_j \\ \text{subject to} \quad & \sum_{j \in N \setminus (I_0 \cup I_1)} w_j x_j \leq c - \sum_{j \in I_1} w_j \\ & x_j = 0 \text{ or } 1, \quad j \in N \setminus (I_0 \cup I_1). \end{aligned}$$

The optimal value to the original problem is $z^* = \max \left\{ z, \hat{z} + \sum_{j \in I_1} p_j \right\}$. The time required by IKR is proportional to the number of variables.

Fayard & Plateau (1982) applied a reduction procedure in their exact algorithm for solving the 0-1KP that slightly differs from IKR only in the way of computing upper bounds for each x_j . The upper bound $\hat{u}_j^0$ and $\hat{u}_j^1$ are defined as

upper bounds for the 0-1 KP, $j \in I_0 \cup I_1$ with additional constraint $x_j = 0$ and $x_j = 1$ respectively.

Dembo & Hammer (1980) proposed a reduction algorithm for the BKP through the Lagrangian approach by choosing the Lagrangian multiplier $\lambda^* = p_b / w_b$. Obviously, the algorithm can also be applied to the 0-1KP. The *Dembo and Hammer reduction algorithm* (DHR) fixes any variable x_j to 1 or 0 when $|p_j - \lambda^* w_j| > UB - LB$ where UB is an upper bound obtained from Lagrangian relaxation and LB is the value of the best known integer solution to the original problem. Their fixing criterion is:

$$x_j = \begin{cases} 1 & , \text{if } p_j - \lambda^* w_j > 0, \\ 0 & , \text{if } p_j - \lambda^* w_j < 0. \end{cases} \quad (2.15)$$

Hence, whenever a better estimation value of the optimal integer solution was known, then more variables would be fixed. However, the authors noted that the value of $\lambda^* = p_b / w_b$ may be not necessarily the best choice depending on which particular variable we wish to fix to 0 or 1.

Balas & Zemel (1980) applied both DHR and IKR reduction methods to fix as many variables as possible. For variables in the core, DHR is first applied to set permanently $x_j = \bar{x}_j$ where $\bar{x}_j = 0$ or 1 when x_j satisfies (2.15) and the lower bound LB is the integer solution value found by a heuristic of the authors. Then IKR is applied to tentatively set for each j in turn, $x_j = 1 - \bar{x}_j$. Finally, the linear program in the remaining variables is solved. If the value of the solution does not exceed LB, then for every j , x_j is permanently set as $x_j = \bar{x}_j$. The reason given by the authors for applying both reduction procedures in this particular order is that IKR is more powerful, but also considerably more expensive computationally, than DHR. Moreover, they presented the more powerful test by exploiting the dominance criterion described below.

Definition (*Dominance criteria*: Balas & Zemel 1980) For $a, b \in N = \{1, 2, \dots, n\}$, if $p_a \geq p_b$ and $w_a \leq w_b$, then x_a dominates x_b .

Clearly, if x_a dominates x_b then we can set $x_b = 0$, if $x_a = 0$ and set $x_a = 1$, if $x_b = 1$.

Martello & Toth (1988) have improved the reduction procedure by using the bound U_2 to compute u_j^0 and u_j^1 and using a binary search to find the break item. By assuming that all items are sorted according to decreasing p_i/w_i , $i = 1, 2, \dots, n$, I_0 and I_1 are determined by finding through binary search for the break item b such that $\bar{w}_{b-1} \leq c < \bar{w}_b$ where $\bar{w}_j = \sum_{i=1}^j w_i$. Initially setting the following values: the lower bound $\bar{z} = \bar{p}_{b-1} = \sum_{i=1}^{b-1} p_i$, and the residual capacity $\bar{c} = c - \bar{w}_{b-1}$. For every $j, s+1 \leq j \leq n$ such that $w_j \leq \bar{c}$, $\bar{z}$ can be improved as $\bar{z} + p_j$. Thus $\bar{c} = \bar{c} - w_j$. Then u_j^0 and u_j^1 are computed as follows:

$$u_j^0 = U'_2 - p_j, \text{ for } 1 \leq j \leq b,$$

$$u_j^1 = U'_2 + p_j, \text{ for } b \leq j \leq n,$$

where U'_2 is the upper bound U_2 with break item b' .

The break item b' is the current break item (for each j) found through binary search as :

$$\bar{w}_{b'-1} \leq c + w_j < \bar{w}_b \text{ for } u_j^0,$$

$$\text{and } \bar{w}_{b'-1} \leq c - w_j < \bar{w}_b \text{ for } u_j^1.$$

Every time u_j^0 is computed, $\bar{c} = c + w_j - \bar{w}_{b'-1}$ and $\bar{z} = \max\{\bar{z}, \bar{p}_{b'-1} - p_j\}$, and similarly for u_j^1 , $\bar{c} = c - w_j - \bar{w}_{b'-1}$ and $\bar{z} = \max\{\bar{z}, \bar{p}_{b'-1} + p_j\}$. Then we have $I_1 = \{j \geq b: u_j^1 \leq \bar{z}\}$ and $I_0 = \{j \leq b: u_j^0 \leq \bar{z}\}$. Generally, this reduction procedure is more effective than IKR since a better lower bound is applied every

time u_j^0 and u_j^1 are computed. For large-sized problems Martello & Toth (1988) apply the reduction procedure with partial sorting only items in an approximate small subset of items or the so-called “*core*” and solve the “*core problem*”.

2.6 Core Problem in Knapsack Algorithms

An interesting and beneficial property of the Knapsack Problem observed by Pisinger (1995b) is that an optimal integer solution may be obtained from the solution of the continuous relaxed problem by changing only a few decision variables around the break item. In addition, sorting items according to non-increasing order of profit-to-weight ratio is generally required for all procedures in solving the problem efficiently. Balas & Zemel (1980) observe that the majority of the computational time for solving large-sized problems is taken by the sorting procedure. On consideration of such property, the authors presented the first algorithm based on the so-called “*core problem*” for efficiently solving the 0-1KP. This has led to numerous developments and to the most computationally successful knapsack algorithms. To avoid a complete sorting, we may consider a sufficiently small subset of the items, called the “*core*” and sort only the items in the core for further procedures. Moreover, the authors have proved that there is a high probability for finding an optimal solution in the core.

The core problem is a Knapsack Problem basically defined on the core. A feasible solution to the core problem is then used as a lower bound in a reduction procedure. Finally, any enumeration method can be applied for solving the core problem in order to obtain an optimal solution to the remaining problem. Suppose, by assuming without loss of generality that $p_j/w_j \geq p_{j+1}/w_{j+1}, j = 1, 2, \dots, n-1$, and the values of ℓ and h such that :

$$\ell = \min\{j : x_j^* = 0\} \quad \text{and} \quad h = \max\{j : x_j^* = 1\}$$

where (x_j^*) is an optimal solution are known, the *core* can be defined as $C = \{\ell, \dots, h\}$. The core problem defined on C as an ordinary Knapsack Problem could be solved by any enumerative algorithm while the variables outside C are set as $x_j^* = 1$ for $j < \ell$ and $x_j^* = 0$ for $j > h$. Definitely, we would never know ℓ and h in advance, but an approximate core may be derived with a tentative value of the core size δ for identifying the break item b without complete sorting. Thus an approximate core can be defined as $C = [b - \delta/2, b + \delta/2]$. Balas & Zemel (1980) proposed the first procedure to approximate a core. Martello & Toth (1988) modified the *Balas-Zemel core procedure* in the algorithm MT2 for solving large-sized 0-1KP. The detail description of “*core procedure*” has been given by Martello & Toth (1988; 1990a, pp.61-67). Some weak points of the *Balas-Zemel core procedure* that make this core problem very hard to solve are observed by Pisinger (1995a; 1995b). Further he has developed the “*findcore procedure*” to approximate a core (see the details in Pisinger & Toth (1998, p.340)). Moreover, several core problems have been developed for efficiently solving large-sized problems in the knapsack family such as the Bounded Knapsack Problem, Multiple-choice Knapsack Problem, Multiple Knapsack Problem, Change-Making Problems, and etc. (Martello & Toth 1990a).

2.7 Exact Algorithms for the 0-1 Knapsack Problem

Among several Knapsack Problems, the 0-1KP is the most important due to its very wide range of applications in the Combinatorial Optimization area. During the last several decades, the 0-1KP has been intensively studied and much progress has been made. Numerous exact algorithms have been developed for the 0-1KP. These are based on: branch and bound or dynamic programming approach. We provide a brief description of each below.

2.7.1 Branch and Bound Algorithms

A number of exact methods for solving the Knapsack Problem using branch and bound approach have been proposed in the last few decades. The differences among these algorithms are in the upper bounding technique, branching strategy, and the dominance criterion in fathoming branches in decision tree. Most effective branch and bound methods are based on a depth-first binary branching scheme to limit the number of branching nodes to $O(n)$ at any stage of the enumeration.

Kolesar (1967) was the first to present an exact solution method for solving 0-1KP through the branch and bound approach. The algorithm searches for a better solution by branching from the feasible node for which the *Dantzig bound* is maximum. This procedure may require large complete memory. Greenberg & Hegerich (1970) have further developed the *Kolesar algorithm* by using depth-first search enumeration. Further improvement has been presented by Horowitz & Sahni (1974). The *Horowitz and Sahni algorithm (HSA)* applied a depth-first search enumeration with the selection of the branching variable the same as in the *Kolesar algorithm*.

HSA mainly consists of two processes: a *forward move* and a *backward move*.

- A *forward move* inserts the largest set of new consecutive items into the current solution subject to the capacity constraint.
- A *backward move* removes the last inserted item from the current solution.

Assume that the items are sorted according to non-increasing profit-to-weight ratio. Then the initial upper bound is obtained from the continuous solution of the 0-1KP by a greedy method and the corresponding integer solution is obtained by setting the break item $x_b = 0$. The algorithm starts from a forward move and tests whether the further forward move could lead to an improved solution by an upper bound test. If so, a new forward move is performed; otherwise, a backward move

follows. At each iteration, the last item has been considered, if the current solution improves the best solution so far then a current best solution is updated. The algorithm stops when no further backtracking can be performed.

Martello & Toth (1977) developed a more efficient branch and bound approach. The main differences between *HSA* and their algorithm **MT1** are (see Martello & Toth 1990a, pp.32-34):

- The upper bound U_2 is used instead of U_1 .
- The forward move is split into two phases: *a building phase* and *a saving phase*. *A building phase* determines the largest set of new consecutive items, which can be inserted into the current solution. *A saving phase* updates the current solution by inserting the set of items assigned by the most recent building phase into the current solution. If the item j is selected, then an upper bound corresponding to the insertion of item j is computed. This upper bound is tested whether it is less than or equal to the value of the best solution so far. If so, the best current solution is updated and a backward move follows. A saving phase is performed only if an upper bound of the new solution does not represent the maximum, which can be obtained by inserting the item j . Otherwise, a backward move is performed.
- A modification, based on the dominance criteria, of the forward procedure is applied. The procedure is performed whenever the current solution could be improved only if the item i is replaced by an item having greater profit and its weight not greater than $w_i + c'$, where c' is the current residual capacity. Obviously, a backward move will be performed whenever no further item $j > i$ fits into the knapsack.
- The upper bounds corresponding to the solution at each node of the decision-tree are computed through a parameter technique based on the storing of information related to the current solution.

The description of the algorithm **MT1** is detailed in (Martello & Toth 1978; Martello *et al.* 1987, pp.222-223 and Martello & Toth 1990a, pp.34-36).

Branch and Bound Procedure with Core Problem

The first branch and bound procedure for solving the 0-1KP on a core problem was presented by Balas & Zemel (1980) and further modified by Fayard & Plateau (1982), Martello & Toth (1988), and Pisinger (1995b).

The algorithm by Balas & Zemel (1980) starts from deriving a core by choosing the core size $\delta = 50$, then determining the continuous solution and the corresponding value by the *Dantzig's method*. The heuristic procedures H and R are applied to the core problem. Procedure H determines an approximate solution based on the dominance criteria. Procedure R is a reduction algorithm that is a modification of DHR and IKR. Finally the enumeration method by Zoltners (1978) is applied to the remaining core problem.

The *Fayard and Plateau algorithm* finds the break item using a procedure similar to that used in the *Balas and Zemel algorithm*. The upper bounds are computed when fixed variables are added. A heuristic solution is determined by utilizing a greedy algorithm (without sorting), then DHR is employed. Finally the remaining problem is enumerated by using a specialized branch and bound algorithm.

The more efficient algorithm MT2 based on solving a core problem has been proposed by Martello & Toth (1988). MT2 is the most general algorithm for solving large-sized 0-1KP. It can be described as follows:

Algorithm MT2 (Pisinger & Toth 1998, p.342)

1. Determine an approximate core $C = [b - \delta/2, b + \delta/2]$ using the same procedure as in the *Balas and Zemel algorithm*. Sort the items in the core and solve the core problem to optimality, deriving a heuristic solution z_C .

2. Derive the enumerative bound U_M . If $U_M \leq \sum_{j=1}^{b-\delta/2-1} p_j + z_C$ then an optimal solution is obtained as the core solution: stop.
3. Apply the *Martello and Toth reduction procedure* (as described in Section 2.5).
4. If all variables $j \notin C$ have been fixed at their optimal value, the core solution is optimal: stop.
5. Sort the items not fixed at their optimal value, and solve the remaining problem to optimality.

Martello & Toth (1988) give a tentative value $\delta = \sqrt{n}$ if $n \geq 100$ and $\delta = n$ if $n < 100$ for the size of an approximate core while Balas & Zemel (1980) suggest the core size $|C| = 50$.

Pisinger (1995a) proposed a new branch and bound algorithm for solving 0-1KP using a different approach from the previous works, called an “*expanding-core algorithm*” containing a main algorithm called the “*primal-dual algorithm*” (*PDA*) (Pisinger & Toth 1998, p.334). *PDA* starts from the break item b in an approximate core $C = [s, t] = [b-1, b]$ and repeatedly inserts an item whenever the current residual capacity is not zero. If the current weight sum exceeds the capacity c , an item will be removed. At each iteration, s and t , $s < b \leq t$, indicates that the variables x_j ; $s < j < t$, are fixed to some value while the remaining variables may be varied arbitrarily, on the assumption that all items $j \leq s$ have been chosen. At any stage the profit sum ($\bar{p}$) and weight sum ($\bar{w}$) corresponding to the current value of the solution vector are determined. Thus if $\bar{w} \leq c$, some item $j \geq t$ must be inserted, and if $\bar{w} > c$, some item $j \leq s$ must be removed from the knapsack. The core is expanded each time the branch and bound algorithm reaches the border of the core. Backtracking is applied when no more items can be inserted or removed, but the *Dantzig bound* U_1 must be relied on to state when backtracking should be performed.

PDA differs from *HSA* since the solution vector cannot be defined directly during the backtracking. Instead, only the item j differing from the break solution is kept in an exception list E each time a better solution has been found. When the algorithm terminates, the solution vector (x_j) is easily defined by first setting $x_j = 1$ for $j = 1, 2, \dots, b-1$ and $x_j = 0$ for $j = b, b+1, \dots, n$, and then changing x_j according to $x_j \in E$.

2.7.2 Dynamic Programming Algorithms

The first dynamic programming based exact algorithm for solving 0-1KP was proposed by Bellman (1954; 1957). The solution technique is based on a recursive formulation of the problem. The *Bellman recursion* is as follows.

For $0 \leq i \leq n$ and $0 \leq \tilde{c} \leq c$, let $f_i(\tilde{c})$ denote the optimal solution value to the subproblem of 0-1KP defined on items $1, 2, \dots, i$ with capacity $\tilde{c}$. The recursion is:

$$f_i(\tilde{c}) = \begin{cases} f_{i-1}(\tilde{c}) & , \text{for } \tilde{c} = 0, \dots, w_i - 1, \\ \max\{f_{i-1}(\tilde{c}), f_{i-1}(\tilde{c} - w_i) + p_i\} & , \text{for } \tilde{c} = w_i, \dots, c, \end{cases}$$

where $f_0(\tilde{c}) = 0$ for $\tilde{c} = 0, \dots, c$. The solution of the above recursion enumerates all states so that each state repeatedly adds a new item to the subproblem. The optimal solution vector may be found by backtracking through the set of states. For a large-sized problem this technique produces a huge amount of states in each iteration that must be saved for backtracking. This leads to the improvement of dynamic programming technique for solving the Knapsack Problem.

Horowitz & Sahni (1974) presented an alternative dynamic programming model. This recursion is based on the decomposition of the original problem of size n into two subproblems of size $n/2$. This is to reduce the number of states in each iteration. A simple dynamic programming is then applied to each

subproblem to enumerate all states. Two sets of states are combined by considering only the pairs of states which produce the largest weight sum not exceeding c at each stage.

Toth (1980) improved the *Bellman recursion* by applying the following dominance criteria to fathom unpromising states:

- (i) if at i^{th} stage, a state has weight sum μ satisfying $\mu < c - \sum_{j=i+1}^n w_j$, then μ

will never reach the value c , thus it can be eliminated,

- (ii) if $c - \min_{i < j \leq n} \{w_j\} < \mu < c$, then the state cannot be improved in the next

iteration, and can be fathomed.

Dynamic Programming with Core Problem

Pisinger (1995b) studied some inconveniences that occur when core problems have been enumerated by using a depth-first branch and bound algorithm. His work establishes that for some cases, degeneration may occur as many items with similar or proportional weights are selected in the core, resulting in difficult problems. Because of the lack of a good lower bound, many algorithms based on the branch and bound approach have difficulty in enumerating the core. To avoid this situation, the author proposed an alternative way for the enumeration, namely a breadth-first dynamic programming approach.

The algorithm *MINKNAP* of Pisinger (1995b) has a property that the enumerated core size is minimal, meaning that the smallest possible symmetrical core can be solved by enumeration. A new dynamic programming recursion is presented based on a property of Knapsack Problems that optimal solutions are found around the fractional variable (break item) of a continuous solution. Thus it is assumed that the items in an approximate core are sorted according to nonincreasing order of profit-to-weight ratios.

Let $f_{s,t}(\tilde{c})$, where $s \leq b$, $t \geq b-1$, $0 \leq \tilde{c} \leq 2c$, be an optimal solution to the problem:

$$f_{s,t}(\tilde{c}) = \max \left\{ \sum_{j=1}^{s-1} p_j + \sum_{j=s}^t p_j x_j : \sum_{j=1}^{s-1} w_j + \sum_{j=s}^t w_j x_j \leq \tilde{c}, x_j \in \{0,1\}, \text{ for } j = s, \dots, t \right\}.$$

The recursion for the enumeration is:

$$f_{s,t}(\tilde{c}) = \max \begin{cases} f_{s,t-1}(\tilde{c}) & , \text{ if } t \geq b, \\ f_{s,t-1}(\tilde{c} - w_t) + p_t & , \text{ if } t \geq b, \tilde{c} - w_t \geq 0, \\ f_{s+1,t}(\tilde{c}) & , \text{ if } s < b, \\ f_{s+1,t}(\tilde{c} - w_s) - p_s & , \text{ if } s < b, \tilde{c} + w_s \leq 2c. \end{cases}$$

with

$$f_{b,b-1}(\tilde{c}) = \begin{cases} -\infty & , \text{ for } \tilde{c} = 0, \dots, \bar{w} - 1; \\ \bar{p} & , \text{ for } \tilde{c} = \bar{w}, \dots, 2c. \end{cases}$$

The recursive enumeration starts at $(s,t) = (b,b-1)$ by either removing an item s from the knapsack, or inserting an item t in the knapsack. An optimal solution to 0-1KP has value $f_{1,n}(c)$. At each stage, since generally far less states than $2c$ need to be considered, the author has modified the algorithm for dynamic programming by reaching. The algorithm *MINKNAP* can be sketched as:

Algorithm *MINKNAP* (Pisinger & Toth 1998, p.344)

1. Find an initial core $C = [b,b]$ by the algorithm “*findcore*”.
2. Run the dynamic programming recursion by reaching, alternately inserting an item t or removing an item s . Check if s, t is within the current core C , otherwise expand the core. Finally, add the remaining items to the core.
3. The process stops when all states in the dynamic programming have been fathomed due to an upper bound test, or all items $j \notin C$ have been fixed at their optimal value.

The core problem is an efficient way of quickly determining a good lower bound; however, it may become inefficient for some difficult instances as investigated by Pisinger (1995b). Recently, Martello, Pisinger & Toth (1997) have proposed the most efficient algorithm for generally solving large-sized 0-1KP based on the core problem. Their algorithm “*combo*” is based on the idea of finding strong bounds by utilizing *cuts* (or a *valid minimal cover inequality* for the 0-1 knapsack polytope) which are incorporated into *MINKNAP* and a modified core problem solved. The cut, called *maximum cardinality constraint*, is added to tighten the problem. Then an optimal surrogate multiplier is determined to relax the cardinality constraint with the knapsack constraint in order to obtain a very strong upper bound. A detail description of *combo* can be found in Martello, Pisinger & Toth (1997). The authors have presented a comparative analysis establishing that *combo* is currently the most efficient exact algorithm for solving 0-1KP.

2.7.3 Approximation algorithms

It is known that all KP’s are *NP-hard*. In spite of this, most exact algorithms based on dynamic programming approach can solve the problems in pseudo-polynomial time with the exception of the *NP-hard* problem in the strong sense. However, some hard problems may not be optimally solvable in a reasonable amount of time. In some situations, such problems may need only a solution that is “sufficiently” close to the optimal.

Sahni (1975) proposed the first polynomial time approximate algorithm. The first fully polynomial time approximation scheme was proposed by Ibarra & Kim (1975). A fully polynomial time approximation scheme cannot be found for the Multiple Knapsack Problem and the Multi-Constrained 0-1Knapsack Problem (Martello and Toth 1990a, pp. 177-178). The basic ideas in the *Ibarra-Kim algorithm* are:

(i) to separate items according to profits into two disjoint sets of “large” items and “small” items;

(ii) to solve the problem through dynamic programming for the large items only, with profits scaled by a suitable scale factor $\delta = \tilde{z}(\epsilon/3)^2$ where $\tilde{z} = \frac{b}{\sum_{i=1}^n p_i}$ and b is the break item as determined in *Dantzig bound* when the nonincreasing order of the profit-to-weight ratios are assumed.

Martello & Toth (see Martello *et al.* 1987, p.233) noted that the time and space complexity of *Ibarra-Kim algorithms* are $O(n \log n + \frac{1}{\epsilon^4} \log \frac{1}{\epsilon})$ and $O(n + \frac{1}{\epsilon^3})$ respectively. Pisinger (1995b, p.24) proved that the time and space complexity of such an algorithm is $O(n/\epsilon^2)$. Further improvements of the fully polynomial time approximation scheme have been obtained by Lawler (1979) and Magazine & Oguz (1981). The most recent modification has been proposed by Capara, *et al.* (1998). The authors have reformulated the problem by adding the maximum cardinality constraint $\sum_{j=1}^n x_j \leq k$. The problem is solved by a new version of a dynamic programming scheme that can be stated as follows:

Let b be an upper bound on the optimal value,

$g_t(a, s)$ for $t = 1, \dots, n$, $s = 1, \dots, k$, $a = 0, \dots, b$, denote the optimal solution value of the following problem :

$$g_t(a, s) = \min \left\{ \sum_{j=1}^t w_j x_j : \sum_{j=1}^t p_j x_j = a; \sum_{j=1}^t x_j = s; x_j \in \{0, 1\}, j = 1, \dots, t \right\}.$$

Initially, $g_0(a, s) = +\infty$ for all $s = 0, \dots, k$; $a = 0, \dots, b$, and then $g_0(0, 0) = 0$. The dynamic programming recursion formula is defined as:

$$g_t(a, s) = \min \begin{cases} g_{t-1}(a, s) \\ g_{t-1}(a - p_t, s - 1) + w_t \end{cases} \quad \text{if } s > 0, a \geq p_t.$$

This algorithm runs in $O(nkb)$ time and requires $O(k^2 b)$ space.

Due to the number of variants in the family of Knapsack Problems, several algorithms have been proposed for solving Knapsack Problems. The book by Martello & Toth (1990a) provides a comprehensive account of the most fundamental Knapsack Problems. Recently, Pisinger (1995b) has proposed a number of very efficient algorithms for solving some Knapsack Problems. A section on Knapsack Problems by Pisinger and Toth (1998, pp.299-428) in Handbook of Combinatorial Optimization Vol.1 gives an excellent overview of Knapsack Problems. In addition, almost all textbooks on Integer Programming and Combinatorial Optimization devote a section to Knapsack Problems, such as Hu (1969), Garfinkel & Nemhauser (1971), Salkin (1975), Sylo, Deo & Kowalik (1983), and Nemhauser & Wolsey (1988). The family of Knapsack Problems has been intensively studied and it will continue to attract the keen interest of researchers.

CHAPTER 3

Algorithms for Some Hard Knapsack Problems

Knapsack problems are typically concerned with selecting from a set of n given items, each with a specified weight and value, a subset of items whose weight sum does not exceed a prescribed capacity and whose value is maximum. This NP-hard problem arises in many applications and has been the focus of considerable research over the past two decades. A number of exact algorithms have been developed for the classical 0-1 Knapsack Problems and its variants. In this chapter, exact algorithms are presented for the following variants of data types: the Subset Sum Problem; the Strongly Correlated 0-1 Knapsack Problem; the Inverse Strongly Correlated 0-1 Knapsack Problem; and the corresponding Bounded Strongly Correlated Knapsack Problem and Bounded Subset Sum Problem.

All our algorithms consist of three stages: the first stage generates an initial solution by a greedy procedure; the second stage refines the approximate solution; and the final stage applies a partial lexicographic search procedure to generate an optimal solution. Extensive computational experiments show that our algorithms are able to solve large problems of size up to one million variables in less than 7 seconds CPU time on a Silicon Graphic Workstation (R 5000) running at a clock speed of 150 MHz. A comparative analysis with some recent effective algorithms is given.

3.1 Introduction

Let $N = \{1, 2, \dots, n\}$ be a given set of items with item i having a weight of w_i and a value (or profit) of p_i , and c the capacity of the given knapsack. The classical **0-1 Knapsack Problem (0-1KP)** can be stated as:

$$\begin{aligned} &\text{maximize} && z = \sum_{i=1}^n p_i x_i \\ &\text{subject to} && \sum_{i=1}^n w_i x_i \leq c, \\ &\text{and} && x_i \in \{0,1\}, \text{ for } i = 1, 2, \dots, n. \end{aligned}$$

The w_i 's and p_i 's are assumed to be positive integers.

As detailed in Chapter 2, 0-1KP's have been extensively studied by many researchers particularly over the past two decades. The excellent book of Martello & Toth (1990a) and the recent survey by Pisinger and Toth (1998, pp.299-428) provides a comprehensive account of the theory and applications of the problem.

As noted in the Chapter 1, data instances which have been used by many researchers in computational experiments concerning the 0-1KP are classified into four types depending on the correlation between profit and weight:

- *Uncorrelated* : there is no correlation between the profit and the weight of an item (p_i and w_i uniformly random in $[1, R]$ where R is the data range).
- *Weakly Correlated* : the profit is (highly)correlated with the weight (w_i uniformly random in $[1, R]$ and p_i uniformly random in $[w_i - R/a, w_i + R/a]$ where a is an integer).
- *Strongly Correlated* : profit is linearly related to the weight of an item (w_i uniformly random in $[1, R]$ and $p_i = w_i + \alpha$ where α is positive integer.) .
- *Inverse Strongly Correlated* : profit is linearly related to the weight of an item (w_i uniformly random in $[1, R]$ and $p_i = w_i + \alpha$ where α is negative integer).
- *Subset Sum* : profit and weight of each item are equal (w_i uniformly random in $[1, R]$ and $p_i = w_i$).

A number of variations to the classical Knapsack Problem have also been studied (Martello & Toth 1990a) including:

- **Bounded Knapsack Problem (BKP)** : here a bounded amount of each item is allowed.

- **Multiple - Choice Knapsack Problem** : here items are to be chosen from disjoint subsets(classes).
- **Multiple - Knapsack Problem** : here several knapsacks are to be filled simultaneously.
- **Multi - Constrained Knapsack Problem** : basically a general integer programming problem.

As noted in Chapter 2 since the early work of Bellman (1954, 1957) and Dantzig (1957), most of the research effort on the 0-1KP has been on developing the methods of branch-and-bound and dynamic programming.

In brief, Branch-and-bound methods for the 0-1KP have been developed over the past three decades with the early contributions made by Gilmore & Gomory (1961), Kolesar (1967), Greenberg & Hegerich (1970), Horowitz & Sahni (1974) and Nauss (1976). This early work was followed by several well-known hybrid algorithms which aimed to improve performance. A reduction procedure was presented by Ingargiola & Korsh (1973). A number of authors Martello & Toth (1977; 1988 & 1997), Müller-Merbach (1978), and Dudzinski & Walukiewicz (1987) modified the upper bound improvement of Dantzig's. In the recent paper of Martello & Toth (1997), 0-1KP's involving all data types with n up to 10000 are solved.

Several effective algorithms for the 0-1KP are hybrids of the dynamic programming method. Toth (1980) presented a dynamic programming algorithm based on modified reduction procedures and Balas & Zemel (1980) solved large 0-1KP's by considering the "*core problem*". The latter approach has given rise to a number of efficient algorithms developed by Fayard & Plateau (1982), Martello & Toth (1988), Pisinger (1995,1997). Recently, Martello, Pisinger & Toth (1997) proposed an efficient algorithm to solve the general 0-1KP with n up to 10000 in less than 0.12 second. Their algorithm, called COMBO, utilizes dynamic programming combined with maximum cardinality of an optimal solution.

The first approximation scheme for the 0-1KP was proposed by Sahni (1975) and a fully polynomial-time approximation scheme was obtained by Ibarra & Kim (1975). A number of authors have examined the implementation of sequential algorithms for the 0-1KP based on both branch-and-bound and dynamic programming. Work on the branch-and-bound was investigated by Lai & Sahni (1984) and further developed by Janakiram *et al.*(1988), Kindervater & Trienekens (1988). More recently the paper of Loots & Smith (1992) details a method that performs both sorting and branch-and-bound steps in parallel. Their algorithm solves 120 problems with n up to 20000 for all data types except the strongly correlated type. Their computational results report average speedups in the range from 1.35 to 1.99, 1.52 to 3.9 and 0.95 to 7.9 for 2, 4, and 8 processors, respectively. The corresponding publication for dynamic programming using a hypercube topology was presented by Lee *et al.*(1988). The recent developments by Lin & Storer (1991) have provided the Connection Machine (CM) technique for parallel algorithms to solve large problems with n up to 800 by using 4096 processors. The CM running time as a function of the number of processors and also as a function of the knapsack capacity were presented graphically. This running time for fixed n increases logarithmically in c , the knapsack capacity. A comparative analysis with the algorithm of Lee *et al.*(1988) demonstrated the superiority of the CM technique.

In this chapter we present specialized algorithms for the 0-1KP and the BKP. We consider only two data types: subset sum type and strongly correlated type for 0-1KP. For the bounded case we refer to the corresponding problems as Bounded Subset Sum Problem (BSSP) and Bounded Strongly Correlated Knapsack Problem (BSCKP). Only these data types are considered since uncorrelated and weakly correlated instances are easy to solve by general algorithms. For the subset sum and the strongly correlated data instances, good approximate solutions can be obtained through the implementation of a greedy algorithm. We present new greedy algorithms which we use in our solution strategy specialized for each type of problem. In brief, our solution strategy involves three stages:

First stage : initialize a solution by a Greedy Algorithm.

Second stage : refine the approximate solution.

Final stage : apply a procedure for optimality.

All our specialized algorithms solve large-sized problems efficiently established by a computational analysis. We consider the Subset Sum Problem (SSP), the Strongly Correlated 0-1KP (SCKP), the Inverse Strongly Correlated 0-1KP and the BKP (BSSP and BSCKP) in Sections 3.3, 3.4, 3.5 and 3.6 respectively. For all cases, the final stage when required (if an optimal solution is not obtained in the previous stages) involves a partial enumeration. Our procedure “*Partial Lexicographic Search*” for this enumeration is common to all problem types and is described in next section.

3.2 Preliminaries

The most effective exact algorithms for solving the 0-1KP utilize the concepts of branch-and-bound or dynamic programming. Usually, the enumeration is curtailed by the use of reduction procedures and upper bound tests. Martello & Toth (1997) and Martello, Pisinger & Toth (1997) propose upper bounds by utilizing additional cardinality constraints. Their upper bounds are determined by solving a relaxation problem as mentioned in Chapter 2. The upper bound test proposed by Pisinger (1995b, pp.69-86; 1997) is used to delete unpromising states after each iteration in a dynamic programming algorithm (*MINKNAP*). This method may lead to a complete enumeration for the Subset Sum Problem (SSP) and the Strongly Correlated 0-1 Knapsack Problem (SCKP).

For example, Pisinger (1995b, pp.69-86; 1997) defined the dynamic programming recursion (*MINKNAP*) as follows:

Let $C = [s, t]$ be the interval of items representing the current core $\{s, s+1, \dots, t\}$ and $b = \min \left\{ h : \sum_{j=1}^h w_j > c \right\}$. Let $f_{s,t}(\tilde{c})$ where $s \leq b, t \geq b-1, 0 \leq \tilde{c} \leq 2c$ be the optimal solution value to the core problem:

$$f_{s,t}(\tilde{c}) = \max \left\{ \sum_{j=1}^{s-1} p_j + \sum_{j=s}^t p_j x_j : \sum_{j=1}^{s-1} w_j + \sum_{j=s}^t w_j x_j \leq \tilde{c}, x_j \in \{0,1\}, \text{ for } j = s, \dots, t \right\}.$$

The recursion for the enumeration is:

$$f_{s,t}(\tilde{c}) = \max \begin{cases} f_{s,t-1}(\tilde{c}) & , \text{ if } t \geq b, \\ f_{s,t-1}(\tilde{c} - w_t) + p_t & , \text{ if } t \geq b, \tilde{c} - w_t \geq 0, \\ f_{s+1,t}(\tilde{c}) & , \text{ if } s < b, \\ f_{s+1,t}(\tilde{c} - w_s) - p_s & , \text{ if } s < b, \tilde{c} + w_s \leq 2c. \end{cases}$$

with

$$f_{b,b-1}(\tilde{c}) = \begin{cases} -\infty & , \text{ for } \tilde{c} = 0, \dots, \bar{w} - 1; \\ \bar{p} & , \text{ for } \tilde{c} = \bar{w}, \dots, 2c. \end{cases}$$

where $\bar{p}$ and $\bar{w}$ are the profit and weight sums of the break solution. The enumeration starts at $[s, t] = [b, b-1]$ and expands by either removing an item s from the knapsack or inserting an item t into the knapsack. An optimal objective function value of 0-1KP is defined as $f_{1,n}(c)$. Each state of dynamic programming model is represented by a triple (π_i, μ_i, v_i) where

$$\pi_i = \sum_{j=1}^{s-1} p_j + \sum_{j=s}^t p_j x_j^*,$$

$$\mu_i = \sum_{j=1}^{s-1} w_j + \sum_{j=s}^t w_j x_j^*,$$

and variable v_i is a representation of the corresponding solution vector $x^* = (x_j^*)$.

The upper bound derived from the next items outside the core is defined as:

$$U(\pi_i, \mu_i) = \begin{cases} \pi_i + (c - \mu_i)p_{t+1} / w_{t+1} & , \text{if } \mu_i \leq c \\ \pi_i + (c - \mu_i)p_{s-1} / w_{s-1} & , \text{if } \mu_i > c. \end{cases}$$

Each state will be fathomed if $U(\pi_i, \mu_i) < z + 1$ where

$$z = \max_{(\pi_i, \mu_i, v_i) \in X_{s,t}} (\pi_i : \mu_i \leq c) ;$$

$X_{s,t}$ denotes the set of states for core $[s, t]$.

Subset Sum Data ($p_i = w_i$)

Here we have

$$f_{s,t-1}(\tilde{c}) = \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t-1} w_j x_j \quad , \quad t \geq b, \quad (3.1)$$

$$f_{s,t-1}(\tilde{c} - w_t) + p_t = \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t-1} w_j x_j + w_t \quad , \quad t \geq b, \tilde{c} - w_t \geq 0, \quad (3.2)$$

$$f_{s+1,t}(\tilde{c}) = \sum_{j=1}^s w_j + \sum_{j=s+1}^t w_j x_j \quad , \quad s < b, \quad (3.3)$$

$$f_{s+1,t}(\tilde{c} + w_s) - p_s = \sum_{j=1}^s w_j + \sum_{j=s+1}^t w_j x_j - w_s \quad , \quad s < b, \tilde{c} + w_s < 2c. \quad (3.4)$$

Obviously, for this type of data $p_j/w_j = 1$, for every j and at state i we have $\pi_i = \mu_i$. Therefore the upper bound $U(\pi_i, \mu_i) = c$ for every i . Hence $U(\pi_i, \mu_i)$ will never be less than $z + 1$ when $z < c$.

Strongly Correlated data ($p_i = w_i + \alpha$)

Here we have

$$f_{s,t-1}(\tilde{c}) = \sum_{j=1}^{s-1} (w_j + \alpha) + \sum_{j=s}^{t-1} (w_j + \alpha) x_j \quad , \quad t \geq b, \quad (3.5)$$

$$f_{s,t-1}(\tilde{c} - w_t) + p_t = \sum_{j=1}^{s-1} (w_j + \alpha) + \sum_{j=s}^{t-1} (w_j + \alpha) x_j + w_t + \alpha \quad , \quad t \geq b, \tilde{c} - w_t \geq 0, \quad (3.6)$$

$$f_{s+1, t}(\tilde{c}) = \sum_{j=1}^s (w_j + \alpha) + \sum_{j=s+1}^t (w_j + \alpha) x_j, \quad s < b, \quad (3.7)$$

$$f_{s+1, t}(\tilde{c} + w_s) - p_s = \sum_{j=1}^s (w_j + \alpha) + \sum_{j=s+1}^t (w_j + \alpha) x_j - w_s - \alpha, \quad s < b, \tilde{c} + w_s < 2c. \quad (3.8)$$

The upper bound $U(\pi_i, \mu_i)$ may be rewritten as

$$U(\pi_i, \mu_i) = \begin{cases} \sum_{j=1}^{s-1} (w_j + \alpha) + \sum_{j=s}^t (w_j + \alpha) x_j^* + \left(c - \sum_{j=1}^{s-1} w_j - \sum_{j=s}^t w_j x_j^* \right) \frac{p_{t+1}}{w_{t+1}}, & \text{if } \mu_i \leq c, \\ \sum_{j=1}^{s-1} (w_j + \alpha) + \sum_{j=s}^t (w_j + \alpha) x_j^* + \left(c - \sum_{j=1}^{s-1} w_j - \sum_{j=s}^t w_j x_j^* \right) \frac{p_{s-1}}{w_{s-1}}, & \text{if } \mu_i > c. \end{cases}$$

It is easy to see that for every state i , $U(\pi_i, \mu_i) = c + k\alpha$ where $k = (s - 1) + \sum_{j=s}^t x_j^*$. Therefore $U(\pi_i, \mu_i)$ will never be less than $z + 1$ when $z = \bar{c} + \bar{k}\alpha$,

where $\bar{c} < c$ and $\bar{k} < k$.

As mentioned in the introduction our algorithms involve three stages. The first two stages produce an approximate solution which often is optimal. If an optimal solution is not obtained in the first two stages the final stage searches for an optimal solution using a *partial lexicographic search (PLS) procedure*. After termination of the second stage, the set of items will be partitioned into a “1-variable set” and a “0-variable set” yielding residual capacity r . If $r > 0$, the PLS procedure is applied to generate subsets of the 1-variable set and subsets of the 0-variable set in order to exchange so as to obtain an improved utilization of the knapsack capacity. More specifically, the procedure is described below:

The Partial Lexicographic Search Procedure (PLS)

Let $I^1 : \{i_j : x_{i_j} = 1, j = 1, 2, \dots, h\}$

where h is an integer $0 < h < n$,

$I_u^1 : \{i_j : 1 \leq j \leq u\}$ where $2 \leq u \leq h$,

P_s : s -ordered subsets of I^1 with the first $s - 1$ elements in I_u^1 ,

$\mathfrak{P}_r : \bigcup_{s=2}^r P_s$,

p_{js} : j^{th} element of P_s ,

$I^0 : \{i_j : x_{i_j} = 0 ; w_{i_j} \leq w_{i_{j+1}}, j = 1, 2, \dots, n-h\}$,

$I_v^0 : \{i_j : 1 \leq j \leq v\}$ where $2 \leq v \leq n-h$,

Q_t : t -ordered subsets of I^0 with the first $t - 1$ elements in I_v^0 ,

$\mathfrak{R}_r : \bigcup_{t=2}^r Q_t$,

q_{mt} : m^{th} element of Q_t ,

z : optimal value,

sum : total weight sum(or profit sum)of 1-variables in second stage,

res : residual capacity in second stage.

Note that $|\mathfrak{P}_r| = \sum_{s=2}^r (h-u) \binom{u}{s-1} + \binom{u}{s}$, $|P_s| = (h-u) \binom{u}{s-1} + \binom{u}{s}$,

and $|\mathfrak{R}_r| = \sum_{t=2}^r (n-h-v) \binom{v}{t-1} + \binom{v}{t}$, $|Q_t| = (n-h-v) \binom{v}{t-1} + \binom{v}{t}$.

Let $J = |P_s|$ and $M = |Q_t|$.

```

input :  $sum, res$  ;
BEGIN
     $kk = 2$  ;  $\ell = 2$  ;
    WHILE  $\ell \leq r$  ;  $kk \leq k^*$  DO
         $u = kk$  ;  $v = kk$  ;
        WHILE  $kk \leq \ell$  DO
            GENERATE all elements of  $P_s = \{ p_{js}, j = 1, 2, \dots, J \}$  ;
            FIND  $VR_j = \sum_{p_{js}} w + res$  , for all  $j$  ;
            FOR  $m = 1$  to  $M$  DO
                GENERATE  $q_{mt}$  (  $m^{th}$  element of  $Q_t$  )
                FIND  $V_{mt} = \sum_{q_{mt}} w$  ;
                FOR  $j = 1$  to  $J$  DO  $D_{jt} = VR_j - V_{mt}$  ;
                IF  $D_{jt} = 0$  THEN
                     $x_a = 0$  , for all  $a \in p_{js}$  ;
                     $x_b = 1$  , for all  $b \in q_{mt}$  ;
                     $z = c$  ;
                    STOP ;
                ELSE
                    FIND  $MD_{kk} = \min_j (D_{jt})$  ;
             $kk = kk + 1$  ;  $\ell = \ell + 1$  ;
        FIND  $minMD_{kk} = \min_{kk} (MD_{kk})$  ;
        BACKTRACK TO FIND the corresponding ordered subset  $p_{js}$ 
        and  $q_{mt}$  of  $minMD_{kk}$  ;
         $x_a = 0$  , for all  $a \in p_{js}$  ;
         $x_b = 1$  , for all  $b \in q_{mt}$  ;
         $z = sum + V_{mt} - VR_j$  ; STOP ;
    END.

```

Experimentally, providing n is not too small, an optimal solution is always obtained by exchanging at most a few items around the item $k = \max_j \{j : \sum_{i=1}^j w_i \leq c\}$. Therefore, in our implementation we restrict the value of two parameters k^* and r to 4. The indices of 1-variables are started counting from item k in descending order until $|I_u^1| = 4$ and similarly the indices of 0-variables are started counting from item $k+1$ in ascending order until $|I_v^0| = 4$.

3.3 Subset Sum Problem (SSP)

Given a knapsack with a capacity c and a set of n items with item i having a weight w_i , the **Subset Sum Problem (SSP)** is to select a subset of items to fill the knapsack such that the total sum of the weights is maximized without exceeding the capacity c . The problem can be written as:

$$\begin{aligned} &\text{maximize} && z = \sum_{i=1}^n w_i x_i \\ &\text{subject to} && \sum_{i=1}^n w_i x_i \leq c, \\ &&& x_i \in \{0,1\}, \quad i = 1, 2, \dots, n. \end{aligned}$$

Without loss of generality, it is assumed that w_i and c are positive integers satisfying $\max_i \{w_i\} < c$ and $\sum_{i=1}^n w_i > c$.

SSP is called the *Value-Independent Knapsack Problem* (Balas & Zemel 1980) or *Stickstacking Problem* (Ahrens & Finke 1975). The most important applications are cargo loading (load a ship with the maximum cargo), cutting stock (find the best way of cutting a portion of space into pieces of different size) and two-processor scheduling (assign n tasks to two processors so as to minimize the finishing time of the entire operation) (Martello & Toth, 1984). Before describing

our algorithm for the SSP, we briefly outline the literature work on this problem. The method of dynamic programming has been extensively used.

The first dynamic programming algorithm for SSP was proposed by Faaland (1973). Ahrens & Finke (1975) proposed a more effective approach with the average time and space reduction to solve the SSP. Martello & Toth (1984) presented a specialized exact algorithm for SSP which is a mixture of dynamic programming and tree search. This method efficiently solves the SSP with weight ordering. Chvatal (1980) has identified SSP as a hard problem when w_i is a randomly chosen integer between 1 and $10^{n/2}$ and $c = \left\lfloor \frac{1}{2} \sum_{i=1}^n w_i \right\rfloor$ since every recursive algorithm (combination of branch-and-bound, dynamic programming and rudimentary arguments) has an exponential time complexity. Pisinger (1995b, pp.157-168) showed that any optimal solution to SSP may be reached through a series of balanced operations applied to dynamic programming with time and space bound of $O(n^2 r)$ where r is the weight bound. The recent exact algorithm, COMBO for general 0-1KP's proposed by Martello, Pisinger & Toth (1997) is also able to solve SSP efficiently.

As an alternative to the above exact dynamic programming based algorithms, fully polynomial-time approximation schemes for SSP have been proposed (Martello & Toth 1990, pp.105-130; Ibarra & Kim 1975 and Lawler 1979). With these schemes any prefixed worst-case performance ratio can be obtained in polynomial time; the time and space complexities grow polynomially in the inverse of the relative error. The schemes are based on the idea of partitioning the items into "*large*" and "*small*" items and solving the problem associated with large items before completing the solution in a greedy way with the small items. Martello & Toth (1984) detailed a worst-case analysis of this polynomial-time approximation scheme for SSP and proved that its worst-case performance is better than that of Johnson's well-known scheme. Fischetti (1990) presented a polynomial-time approximation scheme for SSP requiring linear space. The

scheme is based on a local-search heuristic. Lai (1993) presented a worst-case analysis of greedy algorithms of the SSP with performance ratio $2/3$. The complementary knapsack is defined to collect all items which are not inserted into the knapsack with capacity as the difference of total weight and the capacity of SSP. The algorithm is based on the idea that minimizing the excess of overfilling the complementary knapsack is equivalent to optimally filling, without exceeding the knapsack capacity. Soma *et al.* (1995) presented a variant of the polynomial approximation scheme of Martello & Toth (1984) for SSP. The algorithm has a better experimental error behaviour and comparable running time with the same error. The most recent approximation algorithms for k -items SSP (kSSP) which is a special case of SSP with the additional constraint $\sum_{j \in N} x_j \leq k$; $N = \{1, 2, \dots, n\}$, $1 \leq k \leq n$ have been presented by Capara *et al.* (1998).

A number of other approaches have been studied. Probabilistic results for the SSP were first presented by D'Atri & Puech (1982). Assuming that the weights are independently drawn from a uniform distribution over $\{1, 2, \dots, b\}$ where b is an upper bound on the weight value, they proved that for $b < n$ the simple greedy algorithm can solve the SSP with probability tending to 1 as n goes to infinity. Lagarias & Odlyzko (1983) considered the SSP with equality constraint by assuming that the weights are independently drawn from a uniform distribution over $\{1, 2, \dots, 2^{bn^2}\}$ and the knapsack capacity is the sum of the weights in a randomly chosen subset. The algorithms solve almost all instances for the case $b > 1$. The recent paper of Pferchy (1998) investigated the expected values of some parameter illustrating the behaviour of the same greedy algorithms by using some of the technical tools of D'Atri & Puech.

We now describe our algorithm for SSP which involves three stages. The first stage is based on the same idea of the standard greedy approach of Johnson (1974) and no sorting of the weights is needed. The initial approximate solution is

obtained as follows. Items 1, 2,..., k where $k = \max\{j : \sum_{i=1}^j w_i \leq c\}$ are placed in the knapsack. The second stage attempts to improve the solution. Since in the first stage, the item sequence is randomly ordered, some items from $i = k+1$ to n may have a weight less than or equal to the residual capacity $r = c - \sum_{i=1}^k w_i$ and so an improved solution can be obtained by adding items to the knapsack. After termination of the second stage, a good approximate solution will be obtained. To search for an optimal solution, the PLS procedure is applied in the final stage of the algorithm. The proposed algorithm can be described as follows:

Algorithm CK1

First stage:

```

BEGIN
    sum = 0;
    FOR i = 1 TO n DO
        sum = sum + wi ;
        xi = 1;
        IF sum = c THEN STOP;
        ELSE IF sum > c THEN
            sum = sum - wi ;
            xi = 0 ;
            k = i - 1 ;
        BREAK;

```

Second stage :

```

    r = c - sum ;
    j = k + 1 ;
    WHILE r > 0 and j ≤ n DO
        IF wj ≤ r THEN
            r = r - wj ;
            xj = 1;
            sum = sum + wj ;
            j = j + 1;

```

```

ELSE  $x_j = 0$ ;
IF  $r = 0$  THEN STOP.

```

Final stage :

```

ELSE IF  $r > 0$  THEN call PLS procedure (  $sum, r$  );
END.

```

3.3.1 Computational Results

We implemented our algorithm CK1 in ANSI C on a Silicon Graphic Workstation (SGI R5000) running at clock speed of 150 MHz. To provide a measure of the quality of our algorithm we also ran our test problems on the available code from the authors that implements the COMBO algorithm of Martello, Pisinger & Toth (1997). A comparative analysis based on 900 random problems is carried out. The integer weights w_i 's are randomly generated from a uniform distribution in $[1, R]$ where $R = 1000$ and $R = 10000$. The number of items n varied from 100 to 1000000. We use nine values of n : 100, 300, 1000, 3000, 10000, 30000, 100000, 300000, 1000000. For each n and R we generated 50 problems. The capacity c is selected as $c = 0.5 \sum_{i=1}^n w_i$ (Martello and Toth 1990a, pp.67).

n	R=1000		R=10000	
	CK1	COMBO	CK1	COMBO
100	0.000727	0.002433	0.009326	0.026341
300	0.001230	0.002809	0.004910	0.035066
1000	0.002236	0.002924	0.007666	0.030336
3000	0.001807	0.003451	0.015716	0.025546
10000	0.001321	0.012222	0.022484	0.034772
30000	0.004669	0.040582	0.027919	0.039076
100000	0.015765	0.154293	0.023029	0.154645
300000	0.048148	0.494771	0.049174	0.492164
1000000	0.158000	1.669074	0.160124	-

Table 3.1: Average CPU time (in seconds) of CK1 and COMBO, Silicon Graphic R5000 Workstation.

n	$\frac{z^* - z}{z^*}$		% z optimal	
	R=1000	R=10000	R=1000	R=10000
100	0.000761	0.000747	2	0
300	0.000081	0.000095	10	2
1000	0.000007	0.000006	42	4
3000	0.000000	0.000001	86	18
10000	0.000000	0.000000	100	56
30000	0.000000	0.000000	100	84
≥ 100000	0.000000	0.000000	100	100

Table 3.2 : Performance of first two stages for solving the SSP.

Table 3.1 shows the average computing time, in seconds, of CK1 and COMBO for each set of 50 problems. All 900 generated problems are solved to optimality within an average time of less than 0.17 second by CK1. Clearly, for all cases CK1 is competitive to COMBO. In the table a “-” means that there is not enough memory to run the program on the computer. The results in table 3.1 may be presented by graphically as follows:

Our computational experiments suggested that the solution obtained at the end of Stage 2 is either optimal or close to optimal. Table 3.2 gives the average value

of the statistic $\frac{z^* - z}{z^*}$, where z^* is the optimal value and z is the value of the feasible solution to SSP at the end of stage 2. It also gives the percentage of instances when z is optimal.

3.4 Strongly Correlated Problem (SCKP)

The 0-1KP with strongly correlated data in which profit and weight are linearly related ($p_i = w_i + \alpha$) has been classified as a hard Knapsack Problem (Martello & Toth 1988). The problem can be formulated as:

$$\begin{aligned} \text{maximize} \quad & z = \sum_{i=1}^n w_i x_i + \alpha \sum_{i=1}^n x_i \\ \text{subject to} \quad & \sum_{i=1}^n w_i x_i \leq c, \\ & x_i \in \{0,1\}, \quad i \in N = \{1,2, \dots, n\}, \end{aligned}$$

where the weight w_i , $i \in N$, α and c are positive integers. In order to avoid trivial cases we assume that $\max_i \{w_i\} < c$, $\sum_{i=1}^n w_i > c$.

As algorithms for general 0-1KP's do not efficiently solve the SCKP, a number of specialized algorithms have recently been developed. Pandit & Ravi Kumar (1993) proposed a lexicographic search for SCKP based on the concept of determining a lower bound and an upper bound for the capacity when the weights are in non-decreasing order. This specialized algorithm is able to solve the SCKP with n up to 500 and shows a much better performance than MT2 the well-known algorithm of Martello & Toth (1988). However, it has been observed that the algorithm becomes ineffective when $\alpha = 0$ and n is large. For this case a superior

algorithm (MTH) is given by Martello & Toth (1997) based on branch and bound combined with a partial dynamic programming recursion.

Pisinger proposed, in order to improve performance for hard KP's, the minimal algorithm (MINKNAP)(Pisinger 1997) and a fast algorithm for SCKP's (Pisinger 1998). The MINKNAP algorithm is based on a dynamic programming procedure approached by enumerating a minimal core. The algorithm was extensively tested on all four data types and performed well except for the SCKP. For the SCKP, the algorithm needs to enumerate a large core, in which millions of states may need to be examined. Martello, Pisinger & Toth (1997) proposed the COMBO algorithm which incorporates strong upper bound on the number of possible items in an optimal solution in the dynamic programming process. The dynamic programming part is essentially a refinement of MINKNAP and MTH. Strong upper bounds are obtained through the use of a maximum cardinality constraint. The algorithm was extensively tested and found to perform very well on all four data types. Recently, a specialized effective algorithm for SCKP (Pisinger 1998) was presented. In this approach an additional constraint, derived from an extended cover, is added and then through the use of surrogate relaxation with optimal multipliers the problem is transformed into an equivalent SSP. For pre-processing, a two-optimal heuristic is applied. The algorithm can solve such SSP problems in milliseconds.

We note that in the paper of Pisinger (1998), a two-optimal heuristic is used based on the concept of adding the constraint: $\sum_{i=1}^n x_i = b-1$ where $b = \min\{j: \sum_{i=1}^j w_i > c\}$ when $w_i \leq w_{i+1}$ for $i=1,2,\dots,n-1$. This constraint is utilized to obtain a strong lower bound. The author showed a proof that “an optimal solution x^* to SSP is also an optimal solution to SCKP provided that $\sum_{i=1}^n x_i^* = b-1$ ”. The solution procedure described uses the additional constraint $\sum_{i=1}^n x_i = \beta$ with descending

values of $\beta < b$. The SCKP is formulated as an equivalent SSP using the model of SSP for optimality at the final stage.

We propose a three stage specialized algorithm to solve the SCKP. In the first stage, the items are ordered according to nondecreasing weights. An initial solution is obtained by the same way as the algorithm CK1. This solution has the maximum number of items. In the second stage, the solution is refined by exchanging between items $x_i = 1, i \leq k$ and item $x_j = 0, j > k$. The exchange is based on the criterion that for each iteration we choose the pair (i, j) which gives the minimum value of $w_i + r - w_j$, where r is a current residual capacity. This means that in each iteration the objective value will be increased. The iteration is terminated when $w_j > w_i + r$. After termination of the second stage, if the residual capacity r is positive, the PLS procedure will be applied to search for an optimal solution. The algorithm can be sketched as:

Algorithm CK2

First stage:

BEGIN

$sumw = 0;$

SORT ITEMS IN NON-DECREASING ORDER OF WEIGHTS

FOR $i = 1$ to n DO

$sumw = sumw + w_i ; x_i = 1;$

IF $sumw = c$ THEN STOP; (An optimal solution is obtained)

ELSE IF $sumw > c$ THEN $sumw = sumw - w_i ;$

$x_i = 0;$

$k = i-1 ;$ BREAK ;

Second stage:

$r = c - sumw;$

$maxw = w_k + r;$

$sump = sumw + \alpha k ;$ (comment: $p_i = w_i + \alpha$)

```

IF  $w_{k+1} > maxw$  THEN call PLS procedure( $r, sump$ );
 $s = 1$ ;
FOR  $j = k + 1$  to  $n$  THEN DO
    IF  $x_j = 0$  and  $w_j < maxw$  THEN DO
        FOR  $i = 1$  to  $k$  DO
            IF  $x_i = 1$  and  $w_i = w_j - r$  THEN
                 $x_i = 0$  ;  $x_j = 1$ ;
                 $sump = sump + w_j - w_i$ ;
                STOP ;
            ELSE IF  $x_i = 1$  and  $w_i > w_j - r$  THEN
                FIND  $diff_s = w_i + r - w_j$  ;
         $s = s + 1$ ;
    FIND  $mindiff = \min_s (diff_s)$ 
    BACKTRACK TO FIND THE CORRESPONDING
     $x_m, m \in I_1$  and  $x_t, t \in I_0$  OF  $mindiff$ 
     $x_m = 0$  ;  $x_t = 1$  ;
     $r = r - w_t + w_m$ ;
     $sump = sump + w_t - w_m$  ;
    IF  $r = 0$  THEN STOP ;
     $maxw = w_t + r$ ;

Final stage:
    IF  $r > 0$  THEN call PLS procedure( $r, sump$ );
END.

```

3.4.1 Computational Results

We implemented CK2 in ANSI C on a Silicon Graphic Workstation (SGI R5000) and the following tables present the computational results on the 900 random problems described earlier. We restricted the running time for each problem to 1 second when $n = 100$ and 300. Again we compare our results with COMBO algorithm.

n	$R=1000$		$R=10000$		Average sorting time CK2
	CK2	COMBO	CK2	COMBO	
100	0.000952 ^a	0.014473 ^a (0.015158)*	0.019675 ^b	0.087283 ^b (0.134450)*	0.00015
300	0.000706	0.018304	0.001205	0.073089	0.00051
1000	0.002348	0.024245	0.003129	0.051271	0.00205
3000	0.008289	0.037707	0.008851	0.037001	0.00703
10000	0.030574	0.091637	0.031527	0.089466	0.02721
30000	0.104236	0.224132	0.105222	0.284032	0.09427
100000	0.394602	0.797120	0.397086	1.068551	0.35806
300000	1.369840	2.769327	1.305274	3.274306	1.20310
1000000	4.858109	8.974759	4.820616	-	4.48000

a : The average CPU time for the 49 problems solved optimally.

b : The average CPU time for the 42 problems solved optimally.

* : The average CPU time for all 50 problems.

Table 3.3 : Average CPU time (in seconds) of CK2 and COMBO, Silicon Graphic R5000 Workstation.

n	$\frac{z^* - z}{z^*}$		% z optimal	
	$R=1000$	$R=10000$	$R=1000$	$R=10000$
100	0.000005	0.000000	86	20
300	0.000000	0.000000	100	96
≥ 1000	0.000000	0.000000	100	100

Table 3.4 : Performance of first two stages for solving the SCKP.

Table 3.3 shows the average computing time, in seconds, of CK2 for each set of 50 problems. Of the 900 random problems, 891 are solved optimally within the specified time of 1 second for each problem. In the case of $n = 100$, CK2 with the maximum value of k^* and σ (in PLS procedure) set at 4, cannot solve, in the specified time of 1 second per problem, 1 and 8 problems when $R = 1000$ and 10000, respectively. We observe that when n is greater than 1000, the average computing time of our CK2 does not depend on the value of R since for each n , the computing times between the two values of R are very close. The value in bracket

shows the average CPU time for the 50 problems solved by COMBO for the case in which CK2 cannot solve all 50 problems. We found that an optimal solution for the problems of size $n \leq 100000$ can be obtained within 0.4 second. The bulk of the CPU time in our algorithm is due to sorting.

Table 3.4 gives the average value of the statistic $\frac{z^* - z}{z^*}$, where z^* is the optimal value and z is the value of the feasible solution to SCKP at the end of stage 2. It also gives the percentage of instances when z is optimal.

The results in table 3.3 may be presented graphically as follows:

3.5 Inverse Strongly Correlated Knapsack Problem (ISCKP)

The SCKP may be generalized to the **Inverse Strongly Correlated Knapsack Problem (ISCKP)** which is defined as:

$$\text{maximize} \quad z = \sum_{i=1}^n w_i x_i - \alpha \sum_{i=1}^n x_i$$

$$\text{subject to} \quad \sum_{i=1}^n w_i x_i \leq c,$$

$$x_i \in \{0,1\}, \quad i \in N = \{1, 2, \dots, n\},$$

where the weight w_i , $i \in N$, α and c are positive integers and the weight $w_i > \alpha$.

A specialized algorithm based on lexicographic search was proposed by Pandit & Ravi Kumar (1993) for solving the *Strongly Correlated Knapsack Problem* including both SCKP and ISCKP. The algorithm efficiently solves such problems with n up to 500. Recently Martello, Pisinger & Toth (1997) presented a

comparison of MT2, MINKNAP, MTH and COMBO algorithms for ISCKP and concluded that the COMBO algorithm is competitive.

We propose a two stage specialized algorithm to solve ISCKP. In the first stage, the items are ordered according to nonincreasing weights. An initial solution is obtained by placing item 1, 2,..., k where $k = \max_j \{j : \sum_{i=1}^j w_i \leq c\}$ in the knapsack. This solution has the minimum number of items. If this stage fails to make full capacity, it means that an optimal solution must contain at least k items. Observe that items i and j with $i \leq k$ and $j > k$ cannot be exchanged since $w_j < w_i$ for all j, i . In addition, since we modified PLS procedure to search for an optimal solution by setting $u = k-1$ and $v = k$ in the WHILE loop, an optimal solution is obtained whenever the residual capacity $r \leq \alpha$. The algorithm can be sketched as:

Algorithm CK3

First stage :

BEGIN

$sumw = 0;$

SORT ITEMS IN NON-INCREASING ORDER OF WEIGHTS

FOR $i = 1$ to n DO

$sumw = sumw + w_i$; $x_i = 1;$

IF $sumw = c$ THEN STOP; (An optimal solution is obtained)

ELSE IF $sumw > c$ THEN $sumw = sumw - w_i$;

$x_i = 0;$

$k = i-1$; BREAK ;

Second stage:

$r = c - sumw;$

IF $r \leq \alpha$ THEN STOP; (An optimal solution is obtained)

ELSE IF $r > \alpha$ THEN call PLS procedure (sum, r) ;

(by setting $u = k-1$ and $v = k$)

END.

3.5.1 Computational Results

We implemented CK3 in ANSI C on a Silicon Graphic Workstation (SGI R5000) and the following tables present the computational results on the 900 random problems described earlier. As before, we restrict the maximum value of k^* and σ to 4 and the running time for each problem to 1 second. A comparative analysis with algorithm COMBO is also presented.

n	$R=1000$		$R=10000$	
	CK3	COMBO	CK3	COMBO
100	0.0006973 ^c	0.015787 ^c (0.015133)*	0.003395 ^d	0.091319 ^d (0.110414)*
300	0.001255	0.023907	0.003035	0.067612
1000	0.003201	0.045295	0.008654 ^e	0.058169 ^e (0.801798)*
3000	0.010553	0.034925	0.012395	0.075980
10000	0.039616	0.085456	0.041611	0.111668
30000	0.141803	0.252560	0.143916	0.291328
100000	0.513364	0.857881	0.518675	1.031159
300000	1.705176	2.891316	1.709795	3.519782
1000000	6.067827	9.202884	5.900765	-

c : The average CPU time for the 47 problems solved optimally.

d : The average CPU time for the 45 problems solved optimally.

e : The average CPU time for the 48 problems solved optimally.

* : The average CPU time for all 50 problems.

Table 3.5 : Average CPU time (in seconds) of CK3 and COMBO, Silicon Graphic R500 Workstation.

n	$\frac{z^* - z}{z^*}$	
	$R=1000$	$R=10000$
100	0.014542	0.011837
300	0.005212	0.004416
1000	0.001452	0.001425
3000	0.000508	0.000525
10000	0.000156	0.000150
30000	0.000045	0.000048
100000	0.000014	0.000015
300000	0.000004	0.000004
1000000	0.000001	0.000002

Table 3.6 : Performance of first stage for solving the ISCKP.

Table 3.5 shows the average computing time, in seconds, of CK3 for each set of 50 problems. Of the 900 random problems, 890 are solved optimally. The 10 that are not solved within 1 second by CK3 include: 3, 5 and 2 problems with $(n, R) = (100, 1000)$, $(100, 10000)$ and $(100, 10000)$, respectively. We observe that when n is greater than 10000, the average computing time of CK3 does not depend on the value of R since for each n , the computing times between the two values of R are very close. We found that an optimal solution for problems of size $n \leq 100000$ can be obtained within 0.52 second. The value in brackets gives the average CPU time for 50 problems solved by algorithm COMBO for the cases which CK3 cannot solve all 50 problems.

Table 3.6 gives the average value of the statistic $\frac{z^* - z}{z^*}$, where z^* is the optimal value and z is the value of the feasible solution to ISCKP at the end of Stage 1.

The results in table 3.5 may be presented graphically as follows:

3.6 Bounded Knapsack Problem (BKP)

A generalization of the 0-1KP in which up to b_i items of type i can be selected to fill in a given knapsack capacity c is called the **Bounded Knapsack Problem**. This problem is of course *NP-hard*. Mathematically, BKP can be formulated as:

$$\text{maximize} \quad z = \sum_{i=1}^n p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^n w_i x_i \leq c,$$

$$x_i \in \{0, 1, 2, \dots, b_i\} \text{ and } b_i \text{ is a positive integer, for } i = 1, 2, \dots, n.$$

We have the classical 0-1KP when $b_i = 1$ for each i .

BKP has been studied intensively by only a few researchers. Ingargiola & Korsh (1973) presented a branch-search algorithm combined with a reduction algorithm to solve uncorrelated data type with common upper bound of items. The algorithm is able to solve problems with n up to 500. Martello & Toth (1977)

proposed a branch and bound procedure to solve uncorrelated and weakly correlated data types with up to $n = 100$.

Martello & Toth (1990a, pp.82-84) tackled the BKP by transforming it into a 0-1KP using binary expansion. Each item type i in BKP is expanded to $\tilde{n} = \lceil \log_2(b_i + 1) \rceil$ binary variables in the equivalent 0-1KP. The profits and weights are, respectively, $2^{k-1}p_i$, $2^{k-1}w_i$ for $k < \tilde{n}$ and $(b_i - \sum_{j=1}^{\tilde{n}-1} 2^{j-1})p_i$, $(b_i - \sum_{j=1}^{\tilde{n}-1} 2^{j-1})w_i$ for $k = \tilde{n}$. Now any algorithm for 0-1KP can be applied to the transformed problem. The authors suggest applying the MT2 algorithm. This procedure is effective in solving uncorrelated and weakly correlated types with n up to 50000. However, because of the large number of variables produced by the transformation, computer memory restriction limits the size of the BKP that can be solved. As an alternative, the authors also propose the application of the MT2 heuristic (MT2 is halted at the end of step 2) to the transformed problem. This approximation algorithm can solve all types of data but not efficiently.

Recently, Pisinger (1995b, pp.127-144) modified the MINKNAP algorithm to solve BKP, this modified algorithm is called BOUKNAP. With $R = 1000$ and b_i uniformly chosen from the interval $[5, 10]$, the algorithm can solve almost all types of data with n up to 100000 in less than a second except for the strongly correlated type. This type requires up to 450 seconds computing time for $n = 100000$.

Martello, Pisinger & Toth (1997) classify the Bounded Strongly Correlated Problem as a class of hard problems. The authors showed that BSCKP can be solved by the algorithm COMBO when the problems are transformed to 0-1KP using the technique described in Martello & Toth (1990, pp. 81-102). The BSCKP are generated with w_j uniformly random in $[1, 1000]$, $p_j = w_j + 100$ and the bounds b_j uniformly random in $[1, 10]$. COMBO is able to solve such problems with n up to 10000 in average CPU time 9.99 seconds. The authors suggest that specialized

algorithms for the BSCKP should be developed since the algorithm COMBO is able to solve the similar KP 100 times faster than 0-1KP transformed from the BSCKP.

We modify algorithm CK4 for the BSSP and algorithm CK5 for the BSCKP. The first two stages of the modified algorithm are slightly different from algorithm CK1 and CK2 respectively. If $w_i \leq r$ the item type i will be placed in the knapsack until $w_i > r$ (a current residual capacity) and $x_i < b_i$. The algorithm CK4 and CK5 can be described as:

Algorithm CK4

First stage:

BEGIN

```

    sum = 0; sumb = 0;
    FOR i = 1 TO n DO
        sum = sum + wibi ;
        xi = bi;
        IF sum = c THEN STOP ;    ( An optimal solution is obtained)
        ELSE IF sum > c THEN      sum = sum - wibi ;
                                   xi = 0 ;
                                   k = i ;
                                   BREAK;

```

Second stage :

```

    r = c - sum;
    FOR i = k TO n DO
        xi = 0;
        IF wi = r THEN STOP      ( An optimal solution is obtained)
        ELSE IF wi < r THEN WHILE wi ≤ r and xi < bi DO
                                   r = r - wi; xi = xi + 1;
                                   sum = sum + wi ;

```

IF $r = 0$ THEN STOP ; (An optimal solution is obtained)

Final stage :

ELSE IF $r > 0$ THEN call PLS procedure (sum , r) ;

END.

Algorithm CK5

First stage:

BEGIN

$sumw = 0$; $sumb = 0$; $sump = 0$;

SORT ITEMS IN NON-DECREASING ORDER OF WEIGHTS

FOR $i = 1$ to n DO

$sumw = sumw + w_i b_i$; $x_i = b_i$;

$sumb = sumb + b_i$;

IF $sumw = c$ THEN STOP; (An optimal solution is obtained)

ELSE IF $sumw > c$ THEN $sumw = sumw - w_i b_i$;

$sumb = sumb - b_i$;

$x_i = 0$;

$k = i-1$; $kk = k$;

$r = c - sumw$;

IF $w_i < r$ THEN WHILE $w_i \leq r$ DO

$r = r - w_i$;

$x_i = x_i + 1$;

$sumw = sumw + w_{k+1} x_{k+1}$;

$sumb = sumb + x_{k+1}$;

$kk = k+1$;

$sump = sumw + \alpha sumb$; (comment: $p_i = w_i + \alpha$) BREAK;

IF $sumw = c$ THEN STOP; (An optimal solution is obtained)

Second stage:

$maxw = w_{kk} + r$;

IF $w_{kk+1} > maxw$ THEN call PLS procedure($r, sump$);

$jj = kk$; $s = 1$;

DO

FOR $j = kk + 1$ to n THEN DO

IF $w_j = maxw$ THEN $x_j = x_j + 1$; $x_{kk} = x_{kk} - 1$;

$sump = sump - w_{kk} + w_j$;

STOP ; (An optimal solution is obtained)

IF $x_j = 0$ and $w_j < maxw$ THEN DO

FOR $i = 1$ to $kk+1$ DO

IF $x_i > 0$ and $w_i = w_j - r$ THEN

$x_i = x_i - 1$; $x_j = x_j + 1$;

$sump = sump + w_j - w_i$;

STOP ;

ELSE IF $x_i > 0$ and $w_i > w_j - r$ THEN

FIND $diff_s = w_i + r - w_j$;

$s = s + 1$;

FIND $mindiff = \min_s (diff_s)$

BACKTRACK TO FIND THE CORRESPONDING $x_m, m \in I_1$ and $x_t, t \in I_0$

OF $mindiff$

$x_m = x_m - 1$; $x_t = x_t + 1$;

$r = r - w_t + w_m$;

$sump = sump + w_t - w_m$;

IF $r = 0$ THEN STOP ;

$maxw = w_t + r$;

$jj = t$;

Final stage:

WHILE $r > 0$ DO call PLS procedure($r, sump$);

END.

3.6.1 Computational Results

CK4 and CK5 were implemented in ANSI C on a Silicon Graphic Workstation (SGI R5000) and the following tables present the computational results on the 1350 random problems. The integer weight w_i 's are randomly generated from a uniform distribution in $[1, R]$ where $R = 100$, $R = 1000$, and $R = 10000$. The bounds b_i is randomly distributed in $[1, 50]$ and the capacity $c = 0.5 \sum_{i=1}^n w_i b_i$. We also use the nine values of n as described earlier. As before, we restrict the maximum value of k^* and σ to 4. Further for $n \leq 300$ the running time for each problem restricted to 1 second. To provide a measure of the quality of our algorithms we also ran our test problems on the available code that implements the BOUKNAP algorithm of Pisinger (1995b, pp.127-144).

n	$R=100$		$R=1000$		$R=10000$	
	CK4	BOUKNAP	CK4	BOUKNAP	CK4	BOUKNAP
100	0.000168	0.008644	0.000993	0.133925	0.012307	1.446819
300	0.000170	0.009987	0.000916	0.140345	0.004170	1.202974
1000	0.000173	0.008565	0.001257	0.113895	0.008693	1.293473
3000	0.000512	0.010529	0.001321	0.137553	0.010846	1.321077
10000	0.001944	0.022415	0.018933	0.126295	0.013958	1.394665
30000	0.006145	0.055922	0.000613	0.158654	0.011314	-
100000	0.024928	0.015548	0.026169	0.248373	0.025889	-
300000	0.076332	0.489301	0.077680	-	0.078561	-
1000000	0.260093	-	0.266880	-	0.260672	-

Table 3.7 : Average CPU time (in seconds) of CK4 and BOUKNAP, Silicon Graphic R5000 Workstation.

n	$\frac{z^* - z}{z^*}$			%z optimal		
	$R=100$	$R=1000$	$R=10000$	$R=100$	$R=1000$	$R=10000$
100	0.000007	0.000016	0.000019	78	26	4
300	0.000001	0.000002	0.000002	90	34	4
1000	0.000000	0.000000	0.000000	100	70	12
3000	0.000000	0.000000	0.000000	100	92	34
10000	0.000000	0.000000	0.000000	100	100	70
30000	0.000000	0.000000	0.000000	100	100	96
≥ 100000	0.000000	0.000000	0.000000	100	100	100

Table 3.8 : Performance of first two stages for solving the BSSP

n	$R=100$		$R=1000$		$R=10000$	
	CK5	BOUKNAP	CK5	BOUKNAP	CK5	BOUKNAP
100	0.000181	0.021308	0.000367 ^f	0.290898 ^f (0.279788)*	0.104311 ^g	3.632487 ^g (2.861596)*
300	0.000523	0.090584	0.000579 ^h	1.239531 ^h (1.165229)*	0.006562 ⁱ	12.334515 ⁱ (11.118605)*
1000	0.002045	0.383266	0.002023	4.060636	0.0025615	-
3000	0.007330	1.505939	0.006623	17.72022	0.007698	-
10000	0.052896	7.020092	0.028603	60.828960	0.029219	-
30000	0.109403	-	0.109033	-	0.111188	-
100000	0.413681	-	0.409192	-	0.413800	-
300000	1.367837	-	1.361961	-	1.360962	-
1000000	4.987960	-	4.941349	-	4.889956	-

f : The average CPU time for the 46 problems solved optimally.

g : The average CPU time for the 25 problems solved optimally.

h : The average CPU time for the 47 problems solved optimally.

i : The average CPU time for the 45 problems solved optimally.

* : The average CPU time for all 50 problems.

Table 3.9 : Average CPU time (in seconds) of CK5 and BOUKNAP, Silicon Graphic R5000 Workstation.

n	$\frac{z^* - z}{z^*}$			% z optimal		
	$R=100$	$R=1000$	$R=10000$	$R=100$	$R=1000$	$R=10000$
100	0.000000	0.000000	0.000000	98	84	16
300	0.000000	0.000000	0.000000	100	94	76
1000	0.000000	0.000000	0.000000	98	100	100
≥ 3000	0.000000	0.000000	0.000000	100	100	100

Table 3.10 : Performance of first two stages for solving the BSCKP.

Table 3.7 shows the average computing time, in seconds, of CK4 and BOUKNAP each set of 50 problems. All 1350 generated instances are solved to optimality within an average time of less than 0.27 second by CK4.

The results in table 3.7 may be presented graphically as follows:

Table 3.9 shows the average computing time, in seconds, of CK5 and BOUKNAP for each set of 50 problems. Of 1350 random problems, 1313 are solved optimally. The 37 that are not solved within 1 second by CK5 include: 4, 25, 3 and 5 problems with $(n,R) = (100,1000)$, $(100,10000)$, $(300,1000)$, and $(300, 10000)$, respectively. The value in bracket shows the average CPU time for 50 problems solved by the algorithm BOUKNAP for the case in which CK5 cannot solve all 50 problems. We found that solving by CK5 an optimal solution of the problem size $n \leq 100000$ can be obtained within 0.42 second.

The results in table 3.9 may be presented graphically as follows:

Our computational experiments suggested that the solution obtained at the end of the stage 2 is either optimal or close to optimal. Table 3.8 and Table 3.10 give the average value of the statistic $\frac{Z^* - Z}{Z^*}$, where z^* is the optimal value and z is the value of the feasible solution to BSSP and BSCKP at the end of stage 2. They also give the percentage of problems when z is optimal.

3.7 Conclusions

This chapter has focused on the hard 0-1Knapsack Problem of strongly correlated, subset sum and inverse strongly correlated data instances and the Bounded Knapsack Problem of strongly correlated and subset sum data instances. All our algorithms CK1-5 utilize the PLS Procedure in the search strategy for the optimal solution. Actually, the PLS procedure is designed for the hard 0-1KP as used in CK1-3 but we found that it also works well in solving the BSSP and BSCKP by CK4 and CK5. From our computational experiences, the optimal solutions to most

of the problems considered in this chapter are obtained when the knapsacks are fully filled. However, there are some instances of strongly correlated data (SCKP, ISCKP and BSCKP) that cannot be solved in a reasonable time. These instances may need more enumerating by PLS procedure since their optimal solution has the total weight less than the knapsack capacity. We note that the small problems for this data type are difficult to solve due to increasing the range of random data.

By comparing with the current efficient algorithms for solving general 0-1KP and BKP, CK1-5 all perform very well for all large-sized problems. Especially, CK4 and CK5 can solve problems with up to 1 million variables within 0.3 second for BSSP and less than 5 seconds for BSCKP while BOUKNAP can solve only n up to 300000 for BSSP and n up to 10000 for BSCKP.

CHAPTER 4

Algorithms for Some Hard Unbounded Knapsack Problems

The Unbounded Knapsack Problem is a generalization of the well-known 0-1 Knapsack Problem. Two algorithms for the Unbounded Strongly Correlated Knapsack Problem (algorithm CKU1) and for the Unbounded Subset Sum Problem (algorithm CKU2) are presented in this chapter. These problems seem to be difficult to solve by using dynamic programming or branch and bound algorithms which contain a reduction procedure as a pre-processing stage. All reduction algorithms proposed to date, are not efficient for these problems due to a large number of undominated item types.

Our algorithms do not contain any reduction procedure. They are based on the ideas of exchanging and searching where the number of undominated item types are approximately determined.

An extensive computational analysis shows that both algorithms are able to solve large problems with up to one million variables within less than 0.3 second CPU time on Silicon Graphic Workstation (SGI R 5000) running at clock speed of 150 MHz . A comparative analysis with the well-known algorithm for UKP, MTU2 is also given.

4.1 Introduction

The Unbounded Knapsack Problem (UKP) is a *NP-hard* problem which is a generalization of the well-known 0-1 Knapsack Problem (0-1KP). The problem can be formulated as :

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^n p_i x_i \\ &\text{subject to} && \sum_{i=1}^n w_i x_i \leq c, \\ &&& x_i \geq 0 ; \quad x_i \text{ is integer , } i = 1, 2, \dots, n. \end{aligned}$$

where p_i and w_i are the profit and weight of the item type i respectively, and c is the capacity of the knapsack.

An unlimited number x_i of item type i may be chosen to maximize the total profit but not exceeding the knapsack capacity. The problem can of course be transformed to an equivalent 0-1KP with $\sum_{i=1}^n \lceil \log_2 (\lfloor c / w_i \rfloor + 1) \rceil$ binary variables. Also a transformation into an equivalent Bounded Knapsack Problem (BKP) is possible with each number x_i bounded by $\lfloor c / w_i \rfloor$. The problem has a wide variety of applications, for example in financial management and in industrial problems such as cargo loading, project selection, budget control and capital investment and, etc. In solving a one-dimensional or two-dimensional cutting stock problem

Gilmore & Gomory (1966, 1986) use the knapsack solution to generate the pivot column in the simplex method. In Statistics, Kadane (1968) utilize the Knapsack Problem in the Neymann-Pearson lemma.

The UKP can be transformed to the BKP and the 0-1KP, however Martello & Toth (1990b) noted that algorithms for the BKP and the 0-1KP are not generally efficient for the transformed problems. Exact methods for solving the UKP which are based on branch and bound have been proposed by Gilmore & Gomory (1963), Cabot (1970) and Martello & Toth (1977; 1990b; 1990a, pp.91-103). An efficient algorithm, MTU1, proposed by Martello & Toth (1990b) is derived from the algorithm MT1 for the 0-1KP by replacing the objective upper bound for UKP (see the details of MTU1 and MT1 in Martello & Toth 1990a, pp.91-103). The authors have further developed the algorithm MTU2, a modification of MTU1, and showed that it is the most efficient for solving large-size problems. MTU2 is designed by using the concept of core problem. Assuming that $p_j/w_j \geq p_{j+1}/w_{j+1}$ for $j = 1, 2, \dots, n-1$, the core is defined as $C = \{1, 2, \dots, \bar{n}\}$, where $\bar{n} \equiv \max\{j : x_j > 0\}$. The core problem is formulated as :

$$\begin{aligned} &\text{maximize} && z = \sum_{j \in C} p_j x_j \\ &\text{subject to} && \sum_{j \in C} w_j x_j \leq c, \\ &&& x_j \geq 0 \text{ and integer, } j \in C. \end{aligned}$$

The core size is, of course, approximated by selecting a tentative value for $p_{\bar{n}}/w_{\bar{n}}$. The optimal value will be obtained if the objective function value of the core problem equals to the upper bound. Otherwise, the variables not in the core are reduced and the corresponding core problem is repeatedly solved with a decreased tentative value whenever any variables are left.

The authors compare the Fortran IV implementations of MTU1 and MTU2 with the three standard random data sets: uncorrelated, weakly correlated, and strongly correlated. The comparison shows that MTU2 always dominates MTU1 and can solve very large problems with reasonable computing time even in the case of strongly correlated data, which is a hard instance for MTU1.

Another approach utilizes the concepts of dynamic programming. Gilmore & Gomory (1966) developed specialized dynamic programming algorithms based on the idea of partitioning the knapsack capacity. They prove, assuming that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$, that for any capacity c there exists for each r , $1 \leq r \leq n$, a c_0 such that objective function $z(c) = z_{(1,r)}(c_2) + z_{(r+1,n)}(c_1)$, for some $c_2 \leq c_0$ and $c_1 + c_2 \leq c$. Further, they show that the difference in objective function value of relaxed problem and integer problem of UKP denoted $\theta(c)$ is periodic of period w_1 . The authors propose 4 procedures: *Basis Step-off*, *Ordered Step-off*, *Periodic Step-off* and *Cutting Knife Step-off* for solving the cutting-stock problem modelled as a UKP.

Hu (1969) proposed a dynamic programming algorithm which takes the advantage of the periodic property of $\theta(c)$. Assuming that in the optimum solution $x_i \geq 0$ for all i , then $\theta(c) = \theta(c - w_i)$ plus the loss from not fulfilling the weight w_i with the first item. This loss is defined as $\left(\frac{p_1}{w_1} - \frac{p_i}{w_i} \right) w_i$. Since in the optimal solution, there must be some $x_i > 0$, $\theta(c)$ is obtained as :

$$\theta(c) = \min_i \left\{ \theta(c - w_i) + \left(\frac{p_1}{w_1} - \frac{p_i}{w_i} \right) w_i \right\}.$$

The above gives a recursive formulation in which $\theta(c)$ can be calculated for the variation of knapsack capacity c (see the details in Hu 1969, pp.314-315).

Greenberg (1986) presents two algorithms of dynamic programming which are based on the idea of aggregating the objective function and the constraint. Their algorithms assume that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$ and the greatest common divisor: $\gcd(p_1, p_2, \dots, p_n) = 1$. The author proposes Algorithm 1 to find the solution to UKP by solving the following equivalent problem:

$$F(z) = \min \sum_{i=1}^n (p_i + tw_i)x_i ,$$

subject to $\sum_{i=1}^n p_i x_i \equiv z \pmod{t} ,$

$$x_i \geq 0 , \text{ integer } , i = 1, 2, \dots , n ,$$

where t is first defined as $t = [p_1 c / w_1] + 1$. $F(z)$ can be computed by the dynamic programming recursion :

$$F(z) = \min_i (p_i + tw_i + F(z - p_i)) ,$$

where the arguments of $F(z)$ and $F(z - p_i)$ are taken modulo t .

Algorithm 1 is not suitable for solving the problem when t and c are large. Further cases, a faster method, Algorithm 2 is developed. The UKP is equivalent to the following problem:

$$G(x) = \min \sum_{i=2}^n (w_i - w_1 p_i) p_1 x_i$$

subject to $\sum_{i=2}^n p_i x_i \equiv x \pmod{p_1} ,$

$$x_i \geq 0 , \text{ integer } , i = 2, 3, \dots , n .$$

$G(x)$ is found for $x = z - \lfloor z/c_1 \rfloor c_1$ with $z = p_1 \lfloor c/w_1 \rfloor$ by the dynamic programming recursion $G(x) = \min_{2 \leq i \leq n} \{w_i - w_1 p_i/p_1 + G(x-w_i)\}$, where $G(0) = 0$ and the argument of $G(x)$ (and $G(x - w_i)$) is taken modulo w_1 .

Recently, Andonov, Poirriez & Rajopadhye (1997) have proposed an efficient dynamic programming method for the Unbounded Knapsack problem (EDUK). It works in two phases: the forward phase calculates the optimal value of the profit function and the backtracking phase determines an optimal solution. The authors have developed some fundamental properties of the UKP, namely *dominance relation, periodicity, sparse representation and dominance backtracking*. They incorporated all these properties in the framework of a dynamic programming recurrence to obtain an efficient algorithm.

Algorithms based on branch and bound and dynamic programming utilize a reduction procedure. This is important as the reduction of dominated item types decreases the problem size and thus improves the speed of algorithms. Gilmore & Gomory (1966) proposed the notion of dominance by intuitively selecting the item type which has a smaller weight and larger profit to fill in the knapsack. Martello & Toth (1990a, pp.100-101) introduced a more powerful dominance based on the definition that “*given an instance of UKP, relative to item types set N , item type $k \in N$ is dominated if the optimal solution value does not change when k is removed from N* ”. In addition, they prove that for an item type k , if there exists an item type j such that $\lfloor w_k / w_j \rfloor p_j \geq p_k$, then item k is dominated. By combining with the core problem, all dominated item types can be efficiently eliminated from the core. This technique is exploited in the pre-processing stage of algorithm MTU2.

Dudzinski (1991) showed that the dominance relation given by Martello & Toth (1990a, p.101) is a partial order. Further, he proposed an efficient elimination algorithm for pre-processing techniques with sorting only undominated items by applying the techniques of Polak & Payne (1976) and Majchrzak (1980). Pisinger

(1995b) developed fast reduction algorithms and identified non-trivial data instances for the UKP such that dominance reduction is not efficient. Recently, Andonov, Poirriez & Rajopadhye (1997) formalized the notion of *collective dominance* and *threshold dominance*. The notion of collective dominance is used where an object type is dominated by (a subset of) the other object types, i.e., there is a linear combination of these object types which collectively has a larger profit and lower weight. The notion of threshold dominance occurs when a linear combination of the other object types collectively has a larger profit and lower weight than a finite number instances of the object type. They show that threshold dominance strictly includes collective dominance and enables testing for periodicity. They design the EDUK algorithm that incorporates these properties in an eliminating pre-processing phase. For the strongly correlated data type, their results (in the sense of running time) show that EDUK is better than the standard dynamic programming with preprocessing to remove (simply) dominated item types.

The computational results of all literature algorithms for the UKP show that the strongly correlated data instances seem to be hard. For this data type after reduction, the problems typically still remain large. Pisinger (1995b) considers the so-called Unbounded Subset Sum Problem which is also hard. In this chapter the upper bound for such problems are given in Section 4.2. We present specialized algorithms for solving the Unbounded Strongly Correlated Knapsack Problem in Section 4.3 and for the Unbounded Subset Sum Problem in Section 4.4.

4.2 Upper Bound

By assuming that $p_i/w_i \geq p_{i+1}/w_{i+1}$, $i = 1, 2, \dots, n-1$, Martello & Toth (1990b) proposed the following upper bound for the UKP:

$$U_3 = \max(U^0, \bar{U}^1)$$

where

$$U^0 = \lfloor c/w_1 \rfloor p_1 + \lfloor \bar{c}/w_2 \rfloor p_2 + \lfloor c' p_3 / w_3 \rfloor \quad (4.1)$$

$$\bar{c} \equiv c \pmod{w_1}, \quad c' \equiv \bar{c} \pmod{w_2}, \text{ and}$$

$$\bar{U}^1 = \lfloor c/w_1 \rfloor p_1 + \lfloor \bar{c}/w_2 \rfloor p_2 + \left\lfloor \left(c' + \left\lceil \frac{w_2 - c'}{w_1} \right\rceil w_1 \right) \frac{p_2}{w_2} - \left\lceil \frac{w_2 - c'}{w_1} \right\rceil p_1 \right\rfloor.$$

We propose a stronger upper bound for the Unbounded Strongly Correlated Problem and derive the upper bound for the Unbounded Subset Sum Problem as below:

Strongly Correlated Data ($p_i = w_i + \alpha$)

We observe that the upper bound U_3 is not strong for the strongly correlated data sets. For this type of data the following better upper bound is provided.

$$U_S = c + \lfloor c/w_1 \rfloor \alpha, \quad \text{where} \quad w_1 \leq w_2 \leq \dots \leq w_n.$$

Since the objective function value of the strongly correlated knapsack problem depends on the number of items filled the knapsack capacity, U_S is derived from the idea that the knapsack capacity could be fulfilled with the maximum number of type 1 items. We establish that $U_S \leq U_3$ as follows.

Observe that $\lfloor \bar{c}/w_2 \rfloor = 0$ since $w_1 \leq w_2$. So $c' = \bar{c}$. For the strongly correlated data (4.1) can be written as :

$$\begin{aligned} U^0 &= \lfloor c/w_1 \rfloor (w_1 + \alpha) + \left\lfloor \frac{\bar{c}(w_3 + \alpha)}{w_3} \right\rfloor \\ &= \lfloor c/w_1 \rfloor w_1 + \lfloor c/w_1 \rfloor \alpha + \left\lfloor \bar{c} + \frac{\alpha \bar{c}}{w_3} \right\rfloor \end{aligned}$$

$$\begin{aligned}
&= c - \bar{c} + \lfloor c/w_1 \rfloor \alpha + \left\lfloor \bar{c} + \frac{\alpha \bar{c}}{w_3} \right\rfloor \\
&\geq c - \bar{c} + \lfloor c/w_1 \rfloor \alpha + \bar{c} = U_S .
\end{aligned}$$

Since $U_3 = \max(U^0, \bar{U}^1)$ and $U_S \leq U^0$, then $U_S \leq U_3$ as required .

Subset Sum Data ($p_i = w_i$)

Without sorting, the upper bound for the Subset Sum data is $U_3 = c$. Here $p_i = w_i$ for every i . Hence

$$\begin{aligned}
U^0 &= \lfloor c/w_1 \rfloor p_1 + \lfloor \bar{c}/w_2 \rfloor p_2 + \lfloor c'p_3/w_3 \rfloor \\
&= \lfloor c/w_1 \rfloor w_1 + \lfloor \bar{c}/w_2 \rfloor w_2 + c' .
\end{aligned}$$

Now if $w_2 \geq \bar{c}$ then $\lfloor \bar{c}/w_2 \rfloor = 0$, $c' = \bar{c} - \lfloor \bar{c}/w_2 \rfloor w_2$ and hence

$$U^0 = \lfloor c/w_1 \rfloor w_1 + \bar{c} = c .$$

If, on the other hand, $w_2 < \bar{c}$, then

$$\begin{aligned}
U^0 &= \lfloor c/w_1 \rfloor w_1 + \lfloor \bar{c}/w_2 \rfloor w_2 + \bar{c} - \lfloor \bar{c}/w_2 \rfloor w_2 , \\
&= \lfloor c/w_1 \rfloor w_1 + \bar{c} = c .
\end{aligned}$$

We have thus showed that $U^0 = c$. Further,

$$\begin{aligned}
\bar{U}^1 &= \lfloor c/w_1 \rfloor w_1 + \lfloor \bar{c}/w_2 \rfloor w_2 + \left[\left(c' + \left\lceil \frac{w_2 - c'}{w_1} \right\rceil w_1 \right) - \left\lceil \frac{w_2 - c'}{w_1} \right\rceil w_1 \right] \\
&= \lfloor c/w_1 \rfloor w_1 + \lfloor \bar{c}/w_2 \rfloor w_2 + c' \\
&= U^0 .
\end{aligned}$$

Hence $U_3 = c$.

4.3 The Unbounded Strongly Correlated Knapsack Problem (UNSCK)

The Unbounded Strongly Correlated Knapsack Problem (UNSCK) may be formulated as:

$$\begin{aligned}
 &\text{maximize} && z = \sum_{i=1}^n w_i x_i + \alpha \sum_{i=1}^n x_i \\
 &\text{subject to} && \sum_{i=1}^n w_i x_i \leq c, \\
 &&& x_i \geq 0 \text{ and integer}, i = 1, 2, \dots, n,
 \end{aligned}$$

where the weights w_i , $i = 1, 2, \dots, n$, are positive integers and α is a positive constant.

Assume that the item types are ordered as $\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}}$ ($i = 1, 2, \dots, n-1$). Obviously, the item type 1 has its weight $w_1 = \min_i (w_i)$, since for the strongly correlated data, the relation $p_i/w_i \leq p_{i+1}/w_{i+1}$ is equivalent to $w_i \leq w_{i+1}$. The optimal solution of the LP relaxation of the UKP is $x_1^* = c/w_1$, $x_i^* = 0$ for $i = 2, \dots, n$. It is clear that $x_1 \leq \lfloor c/w_1 \rfloor$ must hold in any integer solution of UKP. This provides the initial feasible integer solution of UKP as $x_1 = \lfloor c/w_1 \rfloor$, $x_i = 0$ for $i = 2, \dots, n$. Therefore, our algorithm for solving the UNSCK is based on the idea of exchanging items of type 1 and the 0-variables $x_i = 0$ such that $w_i < w_{i+1}$, $\forall i \in S - \{x_1\}$ where $S = \{x_i; w_i < w_{i+1}, i = 2, \dots, k-1\}$. Experimentally, all variables $x_i > 0$ of which $w_i < w_{i+1}$ in an optimal solution are obtained when $k < 10$ with the assumption that the weights w_i are distinct among $x_i \in S$.

4.3.1 Dominance Relation

Both Martello & Toth (1990b) and Dudzinski (1991) remarked for general data types that “ *the number of undominated item types is always very small and grows slowly with n* ”. Andonov, Poirriez & Rajopadhye (1997) made the following interesting remark concerning strongly correlated data sets:

“ *For the interval $[10, 10^3]$ none of the corresponding 990 item types is (simply) dominated by any other, whereas the 10 smallest item types are sufficient to dominate all the other item types. If $n \geq 1000$, all distinct item types will be chosen with high probability, and hence, the number of undominated item types will be constant independent of n .*”

However Pisinger (1995b, p.152) comments that “ *this property only holds if the weights are distributed in $[R_1, R_2]$ where R_1 is very small*” (the three references use $R_1 = 10$). He constructs the Strongly correlated data instances as : $p_i = w_i + 100$, $n = 10^5$, $R_1 = 1, 3, 10, 30, 100, 300, 1000, 3000, 10000$ and $R_2 = 30000$ to identify classes of instances which cannot be reduced as efficiently as demonstrated in Martello & Toth (1990b) and Dudzinski (1991). In addition, the following proposition shows that the number of undominated item types for the Strongly correlated data instances depends on not only the lower bound R_1 of the weights but also the constant α such that $p_i = w_i + \alpha$.

Proposition 4.1 Consider the strongly correlated data where $p_i = w_i + \alpha$, then item type i is dominated by item type j if and only if $(k-1)\alpha \geq r$ where $k = \lfloor w_i / w_j \rfloor$ and $r \equiv w_i \pmod{w_j}$.

Proof Let item type i be dominated by item type j , Then

$$\left\lfloor \frac{w_i}{w_j} \right\rfloor (w_j + \alpha) \geq w_i + \alpha, \text{ and so}$$

$$(k-1)\alpha \geq r \quad , \text{ where } k = \lfloor w_i / w_j \rfloor \text{ and } r = w_i - kw_j .$$

Conversely, if $(k-1)\alpha \geq r$ then $k\alpha - \alpha \geq w_i - kw_j$,

$$\text{and so} \quad k(w_j + \alpha) \geq w_i + \alpha .$$

Hence

$$\left\lfloor \frac{w_i}{w_j} \right\rfloor p_j \geq p_i . \quad \blacksquare$$

By the above proposition, clearly for the same interval $[R_1, R_2]$ and n , when $\alpha < w_{min}$ the number of undominated item types is larger than when $\alpha \geq w_{min}$ where w_{min} is the minimum weight. For example, if $p_i = w_i + 5$ and $w_i \in [10, 1000]$, then the weight 10 dominates all item types with weight greater than or equal 30. Also if $p_i = w_i + 10$ or $p_i = w_i + 100$, then the weight 10 dominates all any item types with weight greater than or equal 20 since $r \leq w_{min} - 1$.

4.3.2 Algorithm CKU1 for UNSCK

Our algorithm CKU1 consists of 7 Exchanging Procedures (**Exp**) as follows :

Exp1 : k item(s) of type 1 with k item(s) of type j ,

Exp2 : k items of type 1 with 1 item of type j ,

Exp3 : k items of type 1 with m items of type 2 where $m < k$,

Exp4 : k items of type 1 with ℓ items of type $j > 2$ where $\ell < k$,

Exp5 : k items of type 1 with 1 item of different k types of ascending corresponding weights ,

Exp6 : k items of type 1 with $s \geq 1$ items of different types of descending corresponding weights ,

Exp7 : 1 item of type 1 with 1 item of type j which yields the maximum increased objective function value .

All exchanging procedures are based on the criterion of increasing objective function value when item type 1 is replaced by item type j .

Let the initial value of $k = \min_t \{tw_1 + r \geq w_j\}$ and

$$v = \left\lfloor \frac{kw_1 + r}{w_j} \right\rfloor (w_j + \alpha) - k(w_1 + \alpha) > 0, \text{ where } r \equiv c \pmod{w_1}.$$

In procedure **Exp 4**, the value of k may be increased to improve v by replacing some more items of type j such that the best value of v can be obtained.

The **Exp2**, **Exp3** and **Exp4** are independently applied to the results produced by **Exp1**. The exchange procedure **Exp1**, **Exp5**, **Exp6** and **Exp7** are independent. Each of the procedures **Exp2** – **Exp7** produces a v value independently. All values of v are compared in the final step and the optimal value associated with an optimal solution will be obtained by the procedure that produces the maximum v .

As the majority of the computing time for solving the UKP by MTU1 is spent for sorting item types as $p_i/w_i \geq p_{i+1}/w_{i+1}$, $i = 1, 2, \dots, n-1$. Martello & Toth (1990b) propose the algorithm MTU2 that applies the concept of core problem. The core problem is first presented by Balas & Zemel (1980) to exactly solve the 0-1KP by sorting only a small subset (core) of the items. The UKP solution is obtained by solving the core problem with sorting only the item types in the core. To avoid the complete sorting, we also modify the well-known *quick sort algorithm* to sort only the set of n' smallest weights. The value n' is tentative to assure that S is a subset of the sorted data set. Then the algorithm CKU1 is applied.

Exchanging Procedures

Our initial solution $x_1 = \lfloor c/w_1 \rfloor$, $x_i = 0$, $i = 2, 3, \dots, n$ yields a residual capacity $r = c - x_1w_1$ and an objective function value .

$$z' = \left\lfloor \frac{c}{w_1} \right\rfloor p_1$$

$$= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha < U_s \quad \text{for } r \neq 0.$$

Thus this initial solution may not be optimal for $r \neq 0$.

We now discuss the exchanging procedures.

1. The *ExP1* is based on the idea that by exchanging the same number of item(s) type 1 and item(s) type j leads to an improvement in the objective function value. Moreover, if the capacity c could be fulfilled, an optimal solution is obtained. This is shown by the following propositions:

Proposition 4.2 Suppose $r \equiv c \pmod{w_1} \neq 0$ is divisible by $w_j - w_1$. Then an optimal solution for the UNSCK is obtained by exchanging $k = \frac{r}{w_j - w_1}$ item(s) of type 1 with k item(s) of type j .

Proof Consider the solution :

$$x_1 = \left\lfloor \frac{c}{w_1} \right\rfloor - k, \quad x_j = k, \quad x_\ell = 0 \quad \text{for } \ell \neq 1, \ell \neq j.$$

The objective function value of this solution is :

$$\begin{aligned} z^* &= \left(\left\lfloor \frac{c}{w_1} \right\rfloor - k \right) w_1 + \left(\left\lfloor \frac{c}{w_1} \right\rfloor - k \right) \alpha + kp_j \\ &= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 - kw_1 + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha - k\alpha + kw_j + k\alpha \\ &= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 + k(w_j - w_1) + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha \\ &= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 + kr + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha \end{aligned}$$

$$\text{Hence } z^* = c + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha = U_s. \quad \blacksquare$$

Proposition 4.3 Consider a feasible solution to the UNSCK with $x_i > 0$, $w_i < w_j$ and $kw_j \leq kw_i + r$, where r is the residual capacity. Then exchanging k item(s) of type i with k item(s) of type j leads to an improved feasible solution.

Proof The condition $kw_j \leq kw_i + r$ ensures that replacing k item(s) of type i with k item(s) of type j results in a feasible solution. Clearly the objective function of the new solution changes by an amount $(w_j - w_i)k > 0$. ■

ExP1 may be sketched as :

BEGIN :

Step 1. (1) **for** $i = 2$ **to** n **do**

(2) **if** $w_i > w_1$ **then do** $d_i = w_i - w_1$;

(3) **if** $d_i > r$ **then break** ;

(4) **else** $f_i \equiv r \pmod{d_i}$;

(5) $ff_i = r/d_i$;

(6) **if** $f_i = 0$ **then** an optimal solution is

$$x_1^* = \lfloor c / w_1 \rfloor - ff_i, \quad x_i^* = ff_i.$$

Stop.

else $df_i = ff_i w_1 + r - ff_i w_i$;

find $df_s = \min_i (df_i)$ and

$$df_\ell = \max_i (df_i).$$

Step 2. Use df_s (with exchanging ff_s items of type 1 for ff_s items of type s) and df_ℓ (with exchanging ff_ℓ items of type 1 for ff_ℓ of items type ℓ) as the residual capacity to search for an item type i , $2 \leq i \leq 10$, which $w_i = w_1 + df_s$ or $w_i = w_1 + df_\ell$. If such an item i is found, then the optimal solution is obtained. Otherwise go to Step 3.

Step 3. Repeat Step 1 (line 1-6) by using df_ℓ as r . If an optimal solution cannot be found in Step 3, then go to Step 4.

Step 4. Compute $v = ff_s(w_s - w_1)$. The feasible solution produced by **Exp1** is

$$x_1^* = \lfloor c / w_1 \rfloor - ff_s, \quad x_i^* = ff_s \text{ with residual capacity } df_s.$$

END.

Exp2, **Exp3**, **Exp4** and **Exp6** attempt to improve the objective function value .
All of these procedures are based on the following proposition :

Proposition 4.4 Consider a feasible solution to the UNSCK with $x_1 > 0$, $w_1 < w_j$ and $mw_j \leq kw_1 + r$ where r is the residual capacity. Then exchanging k items of type 1 with m item(s) of type j leads to an improved feasible solution provided that $mw_j > kw_1 + (k-m)\alpha$ when $k > m$.

Proof The condition $mw_j \leq kw_1 + r$ ensures that replacing k items of type 1 with m item(s) of type j results is a feasible solution. Clearly, $mw_j > kw_1 + (k-m)\alpha$ when $k > m$ implies that $mp_j > kp_1$. Thus the objective function of the new solution changes by an amount $mp_j - kp_1$. ■

2. The Exp2 : Remove k items of type 1 to insert 1 item of type j .

Input : output from **Exp1** ($df_s, x_i, i = 1, 2, \dots, 10$)

df_s and x_i are renamed as res and $zz_i, \forall i$ respectively.

The procedure may be sketched as :

BEGIN :

for $i = 2$ to 10 **do**

find the minimum number k_i of item type 1 such that $w_i \leq k_i w_1 + res$.

if $w_i \leq k_i w_1 + res$ **then** $zz_i = zz_i + 1$; $zz_1 = zz_1 - k_i$;

$PR = w_i + \alpha + (zz_1 - k_i)(w_1 + \alpha)$;

if $PR = U_s$ **then** an optimal solution is

found. Stop.

else $dv_i = w_i + \alpha - k_i(w_1 + \alpha)$;

find $dv_m = \max_i (dv_i)$;

$zz_m = zz_m + 1$; $zz_1 = zz_1 - k_m$; **return** ; **END**.

3. The Exp 3: Remove k items of type 1 to insert m item(s) of type 2 where
 $m < k$.

Input : output from **Exp1** ($df_s, x_i, i = 1, 2, \dots, 10$) and
 k_2 from **Exp2**.

The procedure may be sketched as :

BEGIN :

$k = k_2$; $m = 1$;

find the number k of item type 1 and the number m of item type 2 such
that $m(w_2 + \alpha) > k(w_1 + \alpha)$, $k < 10$ where (k, m) gives the minimum residual
capacity.

compute $PU = m(w_2 + \alpha) - k(w_1 + \alpha)$;

$x_2 = x_2 + m$; $x_1 = x_1 - k$;

if $k w_1 + r - m w_2 = 0$ **and** $PU + x_1(w_1 + \alpha) = U_s$ **then** an optimal solution
is obtained, Stop.

else return ,

END.

4. The Exp4 : Remove k item of type 1 to insert ℓ items of type $j > 2$ where
 $\ell < k$.

Input : output from **Exp1** ($df_s, x_i, i = 1, 2, \dots, 10$)

df_s and x_i are renamed as rr and $zx_i, \forall i$ respectively.

The procedure may be sketched as :

BEGIN :

for $i = 3$ **to** 10 **do** the procedure for each item type i similar to item type 2 in the *Exp3* ;

if an optimal solution cannot be found **then** search for item type m which

$$PU_m = \max_i \{ PU_i ; PU_i = \ell_i (w_i + \alpha) - k_i (w_1 + \alpha) \} ;$$

$$zx_m = zx_m + \ell_m ; zx_1 = zx_1 - k_m ; \textbf{return} ;$$

END.

5. The Exp5 is based on the idea from the following proposition .

Proposition 4.5 Suppose that $w_1 < w_{i_1} < w_{i_2} < \dots < w_{i_k}$ and let $r_j = r_{j-1} + w_1 - w_{i_j}$, where r_j is the residual capacity after the j^{th} exchange. Then an optimal solution for UNSCK can be obtained by exchanging k items of type 1 for k different types of items of weights w_{i_h} , $h = 1, 2, \dots, k$, if $r_k = 0$.

Proof Assume that the initial objective function value $z = \left\lfloor \frac{c}{w_1} \right\rfloor (w_1 + \alpha)$.

Let the initial residual capacity be $r_0 = c - \left\lfloor \frac{c}{w_1} \right\rfloor w_1$. The j^{th} exchange will be

done if $w_{i_j} = \min_i (w_i ; w_i < r_{j-1} + w_1)$, where $r_j = r_{j-1} + w_1 - w_{i_j}$.

Let z_j be the corresponding objective function value of the j^{th} exchange.

$$\begin{aligned} z_j &= \left(\left\lfloor \frac{c}{w_1} \right\rfloor - j \right) (w_1 + \alpha) + \sum_{h=1}^j (w_{i_h} + \alpha) \\ &= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 - jw_1 + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha + \sum_{h=1}^j w_{i_h} . \end{aligned}$$

Assuming that k exchanges were done, then

$$\sum_{j=1}^k r_j = \sum_{j=1}^k (r_{j-1} + w_1 - w_{i_j}) .$$

That is

$$r_k + \sum_{j=1}^{k-1} r_j = \sum_{j=1}^k r_{j-1} + kw_1 - \sum_{j=1}^k w_{i_j} .$$

Since $\sum_{j=1}^k r_{j-1} = r_0 + \sum_{j=1}^{k-1} r_j$, thus $r_k = r_0 + kw_1 - \sum_{j=1}^k w_{i_j}$.

If $r_k = 0$, then $\sum_{j=1}^k w_{i_j} - kw_1 = r_0$.

The objective function value

$$\begin{aligned} z_k &= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 - kw_1 + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha + \sum_{h=1}^k w_{i_h} \\ &= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha + r_0 \\ &= \left\lfloor \frac{c}{w_1} \right\rfloor w_1 + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha + c - \left\lfloor \frac{c}{w_1} \right\rfloor w_1 \\ &= c + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha = U_S . \end{aligned}$$

Hence the solution is optimal. ■

Exp5 may be sketched as :

Let $y_i = x_i, i = 1, 2, \dots, n$ (rename the initial solution for this procedure) and

$$wr = w_1 + r .$$

BEGIN :

for $i = 1$ **to** 10 **do**

if $w_i \leq wr$ **then** $r' = wr - w_i$;

$y_1 = y_1 - 1$; $y_i = 1$;

$v' = w_i - w_1$;

if $r' = 0$ **then** an optimal solution is obtained , Stop.

```

else  $wr = w_1 + r'$  ;
for  $j = i + 1$  to 10 do
    if  $w_j = wr$  then an optimal solution is obtained.
        Stop.
    else if  $w_j < wr$  then  $r' = wr - w_j$  ;
         $wr = w_1 + r'$  ;
         $y_1 = y_1 - 1$  ;  $y_i = y_i + 1$  ;
         $v' = v' + w_j - w_1$  ;

```

END.

6. The Exp6 : Remove k items of type 1 to insert $s \geq 1$ items of different types of descending corresponding weights .

Let $xx_i = x_i$, $i = 1, 2, \dots, n$,

$rs = r$ (the initial residual capacity).

Input : k_i , $i = 2, 3, \dots, 10$ produced in **Exp2** and clearly, $k_2 = \min_i (k_i)$

where $w_1 < w_2 < \dots < w_{10}$.

The procedures may be described as follows :

BEGIN :

Step 1. (Try to remove a minimum number of items type 1 to get the possible maximum value of profit by selecting the item type m that gives the maximum increasing value of profit)

Compute $z = xx_1(w_1 + \alpha)$;

$pz = k_2 w_1 + rs$;

for $i = 2$ **to** 10 **do**

find $w_m = \max_i \{ w_i : w_i < p \text{ and } rz = w_1 + p - w_i - w_2 > 0 \}$.

(Since the objective function value depends on the number of items, we try to insert as many as possible. The value rz is computed to select 1 item of

type m to replace k_2 items of type 1 and at least 1 item of type 2 could be added.)

if $m \neq 2$ **then** $xx_1 = xx_1 - k$; $xx_m = xx_m + 1$;

$pz = pz + (w_m + \alpha) - k_2 (w_1 + \alpha)$;

if $m = 2$ **then** $rs = rs + w_1$; $xx_1 = xx_1 - 1$;

$pz = pz - (w_1 + \alpha)$;

(this means that 1 more item of type 1 is needed to be removed for inserting item type $i > 2$)

Step 2 **If** $m = 2$ **then** set $j = 10$;

else $j = m$;

for $i = j - 1$ **to** 2 (step -1) **do**

$pr = z + pz + w_i - w_1$;

(2.1) $r1 = w_1 + rs - w_i$;

if $r1 = 0$ **then** $xx_1 = xx_1 - 1$; $xx_i = xx_i + 1$;

if $pr = U_s$ **then** an optimal solution is obtained

else $pz = pz + w_i - w_1$; **return** ;

(2.2) **if** $r1 \neq 0$ **then** $rz = w_1 + r1 - w_i$; (try again to search for an optimal solution)

if $rz < 0$ **then** the next smaller weight is considered ;

else $rs = r1$;

$xx_1 = xx_1 - 1$; $xx_i = xx_i + 1$;

$pz = pz + w_i - w_1$; $pr = z + pz$;

for $ii = 2$ **to** m **do** repeat procedure (2.1) in Step 2
if an optimal solution is not obtained then remove 1 more item of type 1 and search for an optimal solution.

END.

7. The *ExpP7* searches for the 1 item type j such that $v_j = \max_i \{v_i: v_i = w_i - w_1; w_i < w_1 + r\}$. By the way, if $w_s = w_i + r$ is found then an optimal solution is obtained as $x_1^* = \lfloor c / w_1 \rfloor - 1$, $x_s^* = 1$.

Main Algorithm

The algorithm CKU1 is designed for a large-size UNSCK. The first step is to determine a set $S = \{x_i; w_i < w_{i+1}, i = 2, \dots, k-1\}$ where k is given the value of 10. Of course the modified quick sort algorithm is applied to sort the set of n' smallest weights which contains set S . Then all *ExpP*'s are employed. Finally, all increased objective function values produced by each *ExpP* are compared to determine the maximum value. CKU1 may be described as follows:

Algorithm CKU1

1. Sort n' smallest values such that $w_1 < w_2 < \dots < w_{10}$ are included.
2. Find $x_1 = \lfloor c / w_1 \rfloor$ and initial residual capacity $r = c - \lfloor c / w_1 \rfloor w_1$, if $r = 0$ then **stop** (an optimal solution is found).
3. Rename all items that have the same values as 1 item type to determine set S .
4. Apply *ExpP 1-7*.
5. Compare all increases of objective function value that are returned by each *ExpP* to determine the maximum value corresponding to the optimal value for the UNSCK.

For large-size instances, CKU1 quickly finds an optimal solution by *ExpP1* since all distinct values are randomly generated with high probability in the uniform distribution.

4.3.3 Computational Results

CKU1 has been implemented in ANSI C on a Silicon Graphic Workstation (SGI R5000) running at clock speed of 150 MHz. To provide a measure of the quality of our algorithm, we also ran our test problems on the available code in Fortran IV that implements the algorithm MTU2 of Martello & Toth (1990a, pp.100). A comparative analysis based on 3900 random problems is carried out. The weights w_i are randomly generated from a uniform distribution in $[10, R]$ where $R = 1000$ and $R = 10000$ and $p_i = w_i + \alpha$ given by 3 values of constant α as $\alpha = 10, 100, 1000$. For all problems, the capacity c was set to $0.5 \sum_{i=1}^n w_i$ for CKU1. In the comparative analysis when $n \geq 10000$ in case $\alpha = 1000$ and $n \geq 100000$ in case $\alpha = 100$, c was set as $0.1 \sum_{i=1}^n w_i$ to avoid integer overflows when running MTU2. We use 13 values of n as shown in Table 4.1.

n	$\alpha = 10$		$\alpha = 100$		$\alpha = 1000$	
	CKU1	MTU2	CKU1	MTU2	CKU1	MTU2
100	0.156	1.254	0.154	0.580	0.156	0.559
300	0.191	1.446	0.189	1.24	0.192	1.213
500	0.207	1.777	0.204	2.925	0.205	1.919
1000	0.323	3.054	0.321	3.159	0.321	3.14
3000	0.846	7.529	0.844	8.166	0.844	8.034
5000	1.403	12.722	1.393	13.088	1.389	13.113
10000	2.310	36.632	2.320	40.84	2.345	40.667
30000	6.074	104.213	6.139	132.22	6.267	116.716
50000	10.127	180.031	10.148	227.67	10.134	222.171
100000	25.947	475.539	22.817	524.88	27.154 22.121*	- 505.592*
300000	72.911	1548.457	67.793	2155.798	72.836 67.797*	- 2151.423*
500000	136.542	-	137.038	-	136.530	-
1000000	287.235	-	288.074	-	253.964*	-

* $c = 0.1 \sum_{i=1}^n w_i$

Table 4.1 : Average CPU time(milliseconds) of CKU1 and MTU2 for 50 problems of

UNSCCK on SGI(R 5000), $w_i \in [10,1000]$ and $c = 0.5 \sum_{i=1}^n w_i$.

n	$\alpha=10$		$\alpha=100$		$\alpha=1000$	
	CKU1	MTU2	CKU1	MTU2	CKU1	MTU2
100	0.178	0.709	0.179	0.701	0.182	0.569
300	0.239	1.256	0.237	3.015	0.244	1.183
500	0.264	1.895	0.264	1.913	0.269	1.838
1000	0.343	3.099	0.344	3.087	0.345	3.180
3000	0.854	7.901	0.855	7.78	0.854	7.819
5000	1.206	12.139	1.198	12.712	1.202	12.706
10000	2.130	23.404	2.120	24.466	2.160 2.123*	24.609*
30000	6.218	73.875	6.152	76.699	6.118 6.164*	75.151*
50000	12.38	117.067	12.262	124.787	12.191	-
100000	25.381	264.342	22.969 22.236*	263.366*	25.902	-
300000	73.934	815.003	92.431 93.569*	851.194*	75.646	-
500000	141.901	-	141.45	-	141.908	-
1000000	247.311	-	247.463	-	266.839*	-

$$* c = 0.1 \sum_{i=1}^n w_i$$

Table 4.2 : Average CPU time(milliseconds) of CKU1 and MTU2 for 50 problems of UNSCK on SGI(R 5000) , $w_i \in [10,10000]$ and $c = 0.5 \sum_{i=1}^n w_i$ (and $0.1 \sum_{i=1}^n w_i$).

Table 4.1 and Table 4.2 show the average computing time including the sorting time, in milliseconds of CKU1 and MTU2 for each set of 50 problems with w_i uniformly random in $[10,1000]$ and $[10,10000]$ respectively. CKU1 can solve the UNSCK with n up to 1000000 for all values $\alpha = 10, 100, 1000$ in less than 0.3 second while not enough space to run MTU2 when $n \geq 500000$ where $\alpha = 10, 100$ for both values of R and $n \geq 100000$ where $\alpha = 1000$ in case $R = 1000$ and $n \geq 50000$ where $\alpha = 1000$ in case $R = 10000$. Clearly, for all cases CKU1 is superior to MTU2. In both tables a “-” means that there is not enough memory to run the program on that computer and “*”mean that for such cases the capacity c was set to $0.1 \sum_{i=1}^n w_i$ for the comparative analysis.

The results in table 4.1 and 4.2 may be presented graphically as follows:

Average CPU time(in milliseconds) of CKU1 and MTU2 for $\alpha = 10$, $w_i \in [10,1000]$ and

$$c = 0.5 \sum_{i=1}^n w_i$$

Average CPU time(in milliseconds) of CKU1 and MTU2 for $\alpha = 100$, $w_i \in [10,1000]$

$$\text{and } c = 0.5 \sum_{i=1}^n w_i$$

Average CPU time (milliseconds) of CKU1 and MTU2 for $\alpha = 10$, $w_i \in [10, 10000]$ and

$$c = 0.5 \sum_{i=1}^n w_i$$

Average CPU time (milliseconds) of CKU1 and MTU2 for $\alpha = 100$, $w_i \in [10, 10000]$

$$\text{and } c = 0.5 \sum_{i=1}^n w_i$$

Average CPU time (milliseconds) of CKU1 and MTU2 for $\alpha = 1000$, $w_i = [10, 10000]$ and

$$c = 0.5 \sum_{i=1}^n w_i$$

4.3.4 Hard Instances

As Martello & Toth (1990b) remarked that “*using higher values of w_{min} produces more difficult instances*” and also Pisinger (1995b, pp.147-155) demonstrated that the number of undominated item types is proportional to the w_{min} . Based on these observations Andonov, Poirriez, & Rajopadhye (1997) construct a number of instances in the case of $n = 10^5$, $w_{max} = 10^5$ and $w_i \in [w_{min}, w_{max}]$ with w_{min} varying between 1 and 10^4 . The capacity was fixed to a large value greater than 2×10^8 . They generated the three standard data sets (uncorrelated, weakly correlated and strongly correlated). For the strongly correlated data, p_i was set as $w_i + 100$. All instances are solved by the algorithm EDUK proposed by the authors and compared with MTU2. The computational result was carried out on a 64 bit DEC/Alpha 2100 5/250 machine with EDUK implemented in objective *ocaml* and MTU2 in FORTRAN 77. The results show that both algorithms are fast for only

uncorrelated data. The algorithm EDUK has better performance than MTU2 for all types of data. However, in the case of strongly correlated data the average running time when $w_{min} \geq 5000$ shows the difficulty of the problems for EDUK. We note that in some cases EDUK required more than 100 seconds and more than 500 seconds when $w_{min} = 5000$ and $w_{min} = 10000$, respectively.

We also test these hard instances (the strongly correlated data sets) by our algorithm CKU1. All problems are generated as above and the capacity c is generated as $c = 2 \times 10^8 + k$ where k is uniformly random in $(0, 10^6)$. The following table shows the result.

w_{min}	CKU1
1	24.201
3	23.336
10	24.091
30	24.170
100	23.279
300	25.233
1000	24.955
3000	23.089
5000	25.867
7000	23.025
9000	22.634
10000	23.520

Table 4.3 : Average CPU time (milliseconds) of CKU1 for the UNSCK 50 problems with $n = 10^5$, $R = 10^5$, $p_i = w_i + 100$, c is uniformly random in $(2 \times 10^8, 2 \times 10^8 + 10^6)$, and w_i is random in (w_{min}, R) .

We tried to solve the same sets of instances by MTU2 on the same machine. However, after 2 hours, MTU2 has not shown any result for any case. We observe that the average running time of CKU1 is quite stable in case of $n = 10^5$, $w_{max} = 10^5$ and $\alpha = 100$ (as shown in Table 4.2). This means that the values of w_{min} and capacity do not affect CKU1.

4.4 Unbounded Subset Sum Problem (UNSSP)

The Unbounded Subset Sum Problem (UNSSP) may be formulated as :

$$\begin{aligned} \text{maximize} \quad & z = \sum_{i=1}^n w_i x_i \\ \text{subject to} \quad & \sum_{i=1}^n w_i x_i \leq c, \\ & x_i \geq 0 \text{ and integer, } i = 1, 2, \dots, n, \end{aligned}$$

where the weights $w_i, i = 1, 2, \dots, n$ are positive integers.

To our knowledge, there is no empirical evidence in the literature of UKP for solving the UNSSP. Only Pisinger (1995b) considers this problem in a comparative analysis of his reduction algorithm with some literature reduction algorithms. The results show that for all testing, reduction algorithms do not work well for the UNSSP when it is compared with the problems of other data types due to a large number of undominated item types. By Proposition 4.1, we note that for the UNSSP the item type i is dominated by item type j if and only if w_i is divisible by w_j (since if $\alpha = 0$ then $(k-1) \geq r$ only when $r = 0$).

We propose the algorithm CKU2 for solving the UNSSP. CKU2 is modified from our algorithm CK4 for solving the Bounded Subset Sum Problem. CKU2 contains 3 steps: determine an initial solution, improve the objective function value and finally search for an optimal solution. It may be described as follows :

Algorithm CKU2

BEGIN

Step 1. Let $n^* = \max(100, n/100)$;

for $i = 1$ *to* n *do* $x_i = 0$;

for $i = 1$ *to* n^* *do* $k_i = c \pmod{w_i}$;

$$b_i = \lfloor c / w_i \rfloor ;$$

if $k_i = 0$ **then** an optimal solution is obtained, **Stop**.

else find $b_m = \min_i (b_i) ;$

$$sumw = 0 ;$$

for $i = 1$ **to** $n/4$ **do** $sumw = sumw + b_m w_i ;$

$$x_i = b_m ;$$

if $sumw = c$ **then** an optimal solution is obtained,

Stop.

else if $sumw > c$ **then while** $sumw > c$ **do**

$$sumw = sumw - w_i ;$$

$$x_i = x_i - 1 ;$$

$$s = i ;$$

$$r = c - sumw ;$$

Step 2. (Each iteration , find w_i that is less than the current residual capacity)

(1) **for** $i = 1$ **to** n **do**

(2) **if** $w_i \leq r$ **then** $r' = r \pmod{w_i} ;$

(3) **if** $r' = 0$ **then** an optimal solution is found, **Stop**.

(4) **else** $r = r - w_i ;$

(5) $sumw = sumw + w_i ;$

(6) $x_i = x_i + 1 ;$

(7) $m = i ;$

else $r = r + w_s ;$

$$x_s = x_s - 1 ;$$

repeat procedure (1) - (7) **until** m is found ;

for $i = 1$ **to** m **do**

if $x_i > 0$ **then** $rw_i = w_i + r ;$

for $j = 0$ **to** n **do if** $j \neq i$ **then**

if $w_j = rw_i$ **then** an

optimal solution is found . **Stop**.

if an optimal solution cannot be found in the previous steps *then* call
 PLS procedure (the detail description of this is given in Section 3.2).
END.

4.4.1 Computational Results

CKU2 has been implemented in ANSI C on a Silicon Graphic Workstation (SGI R5000) running at clock speed of 150 MHz . For a comparative analysis we also ran our test problems on the available code in Fortran IV that implements MTU2 of Martello & Toth (1990). The computational experiment based on 1300 random problems where the integer weights are randomly generated from a uniform distribution in $[10, R]$ with $R = 1000$ and $R = 10000$. The capacity c is fixed at $0.5 \sum_{i=1}^n w_i$. We use 13 values of n and generate 50 problems in each case as shown in Table 4.4.

n	$R=1000$		$R=10000$	
	CKU2	MTU2	CKU2	MTU2
100	0.379	3.734	14.046	6.938
300	0.341	4.053	4.762	7.152
500	0.304	4.362	4.509	7.630
1000	0.287	5.098	3.104	9.611
3000	0.629	8.697	2.144	13.432
5000	0.620	12.725	2.375	16.238
10000	1.367	22.427	2.866	24.912
30000	3.729	77.351	4.265	96.612
50000	4.937	133.221	7.170	189.224
100000	9.366	279.592	12.443	464.355
300000	8.768	871.267	26.135	1917.478
500000	23.814	1498.987	14.245	-
1000000	46.100	-	178.604	-

**Table 4.4 : Average CPU time(milliseconds) of CKU2 and MTU2 for Unbounded Subset
 Sum Problem (50 problems) running on SGI(R 5000) with $c = 0.5 \sum_{i=1}^n w_i$.**

Table 4.4 gives the average CPU times in millisecond(s) of CKU2 and MTU2 for each set of 50 problems. CKU2 can solve the UNSSP to optimality with n up to 1000000 within less than 50 milliseconds while MTU2 can solve the problems with n up to $n = 500000$ within 150 milliseconds where $w_i \in [10,1000]$. In case $w_i \in [10,10000]$, CKU2 can solve the UNSSP to optimality with n up to 1000000 within less than 200 milliseconds while MTU2 can solve the problems with n up to $n = 300000$ within 2000 milliseconds. These results show that CKU2 is faster than MTU2 for all cases except in case $n = 100$ and $R = 10000$. In the table a “—” means that there is not enough memory to run the program on the computer.

The results in table 4.4 may be presented graphically as follows:

4.5 Conclusions

This chapter proposes the specialized algorithms, CKU1 and CKU2 for the Unbounded Strongly Correlated Knapsack Problem and the Subset Sum Problem. Both algorithms are fast since no reduction procedure is required. As previously shown by Proposition 4.1 for the UNSCK, we can approximately determine the number of undominated item types. Then the problem of undominated item types is solved with the strong upper bound of the objective function value U_S . Even though the value of w_{min} is increased, if n is sufficiently large, the distinct weights will be uniformly random with high probability in range $[w_{min}, R]$. For the large problems, the procedure *Exp1* or *Exp5* would easily obtain an optimal solution. Moreover, the different capacity values do not effect the running time of CKU1 as shown in Section 4.3.4. Experimentally, all test problems are solved to optimality. The algorithm CKU2, the modification of our algorithm CK4 for the Bounded Subset Sum Problem, works well for solving the UNSSP and generally performs

better in most instances than the MTU2 algorithm of Martello & Toth (1990a, p.100).

CHAPTER 5

Exact Algorithms for Change-Making Problems

The Change-Making problem is a particular type of single Knapsack Problem. The name comes from considering the problem of a cashier making an amount of change using the least possible number of coins of different denominations. The problem is *NP-hard*.

We propose exact algorithms for the Unbounded Change-Making Problem, UCMP, (algorithm CKUC) and for the Bounded Change-Making Problem, BCMP, (algorithm CKBC). A combinatorial algorithm NEXCOM for generating all compositions of a positive integer m into k parts is utilized in the search strategy to obtain an optimal solution. When the item types are randomly generated in the range $R = 10^3$ and $R = 10^4$, CKUC can solve large-size UCMP's with up to $n = 300000$ within 51 milliseconds and CKBC can solve the BCMP's, with up to $n =$

30000 within 3.5 seconds. Comparative analysis for some variants of problems is also given.

5.1 Introduction and background

The Change-Making Problem (CMP) or the so-called the *Coin-Changing Problem* (Tien & Hu, 1977) arises when a cashier needs to provide a given amount of change by using the least number of coins. It is assumed that the number of coins of each denomination is unlimited. Given n coin denominations with values w_i , $i = 1, 2, \dots, n$, we consider the problem of finding the minimum number of coins needed to make an amount of change c . The problem can be applied to some classes of unidimensional cargo-loading and cutting stock problems (Martello & Toth, 1990a, p.138). The CMP has also been used with linear programming in the solution of the trim-loss-cutting problem (Wright, 1975).

Consider the well-known knapsack problem in minimization form with a single capacity constraint:

$$\begin{aligned} &\text{minimize} && \sum_{j=1}^n p_j x_j \\ &\text{subject to} && \sum_{j=1}^n w_j x_j \geq c, \\ &&& x_j \geq 0 \text{ and integer}, j = 1, 2, \dots, n. \end{aligned}$$

It is assumed, without loss of generality, that the w_j 's and c are positive integers and $c > w_j$ for every j .

The CMP is a particular case of the above problem.

$$\begin{aligned}
\text{CMP1 :} \quad & \text{minimize} \quad \sum_{j=1}^n x_j \\
& \text{subject to} \quad \sum_{j=1}^n w_j x_j = c, \\
& \quad \quad \quad x_j \geq 0 \text{ and integer, } j = 1, 2, \dots, n.
\end{aligned}$$

This problem is *NP-hard* (Luker 1975). In the literature, CMP is generally solved with the assumption that $w_1 > w_2 > \dots > w_n$. Note that, the condition $w_n = 1$ may be added into the assumption to guarantee that there exists a feasible solution for every positive integer c due to the equality constraint.

Chang & Gill (1970) propose a recursive algorithm for exactly solving the problem by taking $w_n = 1$. A greedy algorithm, called “*the standard algorithm*” (or “GREEDYUM” in (Martello & Toth 1990a, p.141)), for finding a feasible solution to the CMP is also presented. GREEDYUM is a procedure that places items in the knapsack according to largest weight. For example, when $n = 4$, $w_1 = 12$, $w_2 = 7$, $w_3 = 5$, $w_4 = 1$ and $c = 22$, the greedy algorithm yields the solution vector $(1, 1, 0, 3)$ which has the objective function value of 5. Note that for this example, the optimal solution value is 3 corresponding to the solution vector $(1, 0, 2, 0)$. In the case when $c = 20$, then the greedy algorithm produces the optimal solution $(1, 1, 0, 1)$.

A number of tests for checking the optimality of greedy solutions have been developed. Magazine, Nemhauser & Trotter (1975) propose some tests based on a theorem known as the “*one-point theorem*”, providing necessary and sufficient conditions. The authors also presented an error analysis when the greedy algorithm fails to produce an optimal solution. Chang & Korsh (1976) develop an efficient test to determine whether all greedy solutions are optimal. Their tests are an extension of previous tests in the literature. Furthermore, the problem of maximum percentage error when the greedy solution does not work has been extensively investigated by Tien & Hu (1977).

For solving the CMP two exact algorithms have been proposed: one based on dynamic programming and another based on branch and bound.

▪ **Dynamic Programming approach** (Martello & Toth, 1990a, pp.145-146):

Let $f_k(\hat{c})$ be the optimal solution corresponding to the sub-instance of CMP consisting of item types $1, \dots, k$ and capacity $\hat{c}$ where $k = 1, \dots, n$ and $\hat{c} = 0, 1, \dots, c$. ($f_k(\hat{c}) = \infty$ if no solution of value $\hat{c}$ exists). Then we have,

$$f_1(\hat{c}) = \begin{cases} \infty & , \text{for all positive } \hat{c} \leq c \text{ such that } \hat{c}(\bmod w_1) \neq 0; \\ s & , \text{for } \hat{c} = sw_1, \text{ with } s = 0, \dots, \lfloor c/w_1 \rfloor. \end{cases}$$

Consider the increasing values of k from 2 to n and, for each k , increasing values of $\hat{c}$ from 0 to c , $f_k(\hat{c})$ can be determined as :

$$f_k(\hat{c}) = \min \{ f_{k-1}(\hat{c} - sw_k) + s : 0 \leq s \leq \lfloor \hat{c}/w_k \rfloor \}$$

where s is an integer. The optimal solution value of the CMP is then given by $f_n(c)$. To reduce the time complexity, the improved recursion is obtained as:

$$f_k(\hat{c}) = \begin{cases} f_{k-1}(\hat{c}) & , \text{for } \hat{c} = 0, \dots, w_k - 1, \\ \min(f_{k-1}(\hat{c}), f_k(\hat{c} - w_k) + 1) & , \text{for } \hat{c} = w_k, \dots, c. \end{cases}$$

Wright (1975) has noted that, if the item types are sorted as: $w_1 < w_2 < \dots < w_n$, at each stage k , only values of $\hat{c}$ not greater than $w_k w_{k+1}$ need to be considered. Since the weight sum of w_k items of type $k+1$ is equal to the weight sum of w_{k+1} items of type k and the previous one yields less number of items. To improve the efficiency, the author suggests that instead of considering $\hat{c}$ from 0 to c in each of the n stages, the procedure can be reversed by considering k from 1 to n for each

value of $\hat{c}$. The optimal value of f^* is obtained from the equation: $f^*(\hat{c}) = \min_{\hat{c}-w_i > 0} f^*(\hat{c} - w_i) + 1, i = 1, 2, \dots, n$.

▪ **Branch and bound approach:** A depth-first search procedure for solving the CMP was first proposed by Martello & Toth (1990a, pp.146-149). The algorithm called CMP2 (Martello & Toth, 1980) or MTC1 (Martello & Toth, 1990a, pp.147-148) is developed by making use of a dominance criterion. An extensive comparative analysis establishes the superiority of MTC1 over the Chang-Gill algorithm and Wright algorithm. For solving large-sized problems, Martello & Toth (1990a, pp.149-152) have developed a more efficient algorithm MTC2. The algorithm obtains an optimal solution by solving a core problem. The core is defined as $C = \{j_1, j_2, j_3, \dots, j_{n^*}\}$ with j_1, j_2, j_3 being the three item types of the largest weight and $j_4, \dots, j_{n^*}$ the other n^*-3 item types. To determine the lower bound, the 3-largest weights are needed. The expected core size is $n^* \leq n-3$, the authors suggested that a good value for n^* is $\min \left(n, \max \left(500, \left\lfloor \frac{n}{20} \right\rfloor \right) \right)$. The core problem can be stated as:

$$\begin{aligned} &\text{minimize} && \sum_{j \in C} x_j \\ &\text{subject to} && \sum_{j \in C} w_j x_j = c, \\ &&& x_j \geq 0 \text{ and integer, } j \in C. \end{aligned}$$

MTC2 solves the core problem by utilizing MTC1. The description of MTC1 and MTC2 are given in (Martello & Toth 1990a, pp.147-150). The authors have presented a comparative analysis of MTC2 and MTC1 and the computational results show that MTC2 is superior to MTC1. When $n \geq 1000$, the core problem is built and MTC2 is applied. MTC2 can solve for $R = 10^5$ large-sized problems with $n = 20000$; MTC1 could not solve problems in reasonable time when $n > 5000$.

Throughout this chapter we consider two types of CMP's. The **Unbounded Change-Making Problem (UCMP)** in which there are an unlimited number of available items of each type and the **Bounded Change-Making Problem (BCMP)** in which the maximum number of each type of item is specified. The UCMP can be considered as a particular case of BCMP where the maximum number b_j of item type j is set as $\lfloor c/w_j \rfloor$. The greedy algorithm for BCMP and the algorithm MTCB are derived from the procedures GREEDYUM and MTC1, respectively, by replacing the maximum number of items for each type as $x_j = \min(\lfloor \bar{c}/w_j \rfloor, b_j)$. To our knowledge, only Martello and Toth (1990a, pp.153-156) have studied the BCMP. They remarked that the solution of BCMP appears harder than that of UCMP and the value of bound b_j did not alter the difficulty.

In this chapter, we propose exact algorithms for solving the Change-Making problems: the UCMP and the BCMP. In our algorithms, we utilize the algorithm NEXCOM (Nijenhuis & Wilf, 1975) for generating all compositions of a given number into a specific number of parts. The description of the algorithm NEXCOM is given in the next section. We consider the UCMP and the BCMP in Sections 5.3 and 5.4 respectively. The comparative analysis for each algorithm is also given.

5.2 Algorithm NEXCOM

Let m and k be fixed positive integers. A composition of m into k parts is defined as a representation of the form $m = p_1 + p_2 + \dots + p_k$ in which $p_i \geq 0$ ($i = 1, 2, \dots, k$) and the order of the summands is important. For example, there are exactly 15 compositions of $m = 4$ into $k = 3$ parts, namely,

$$\begin{aligned} 4 &= 4 + 0 + 0 = 1 + 3 + 0 = 2 + 1 + 1 = 2 + 0 + 2 = 1 + 0 + 3 \\ &= 3 + 1 + 0 = 0 + 4 + 0 = 1 + 2 + 1 = 1 + 1 + 2 = 0 + 1 + 3 \\ &= 2 + 2 + 0 = 3 + 0 + 1 = 0 + 3 + 1 = 0 + 2 + 2 = 0 + 0 + 4. \end{aligned}$$

The algorithm NEXCOM is an algorithm for generating all compositions of m into k parts. Let a composition of m into k parts denote $(p_i)_k$ where $i = 1, \dots, k$. NEXCOM can be described as:

Algorithm NEXCOM (Nijenhuis and Wilf, 1975)

NEXCOM(m, t)

input : m, t ;

output : (p_i) ;

(1) [First entry] $p_1 \leftarrow m$; $p_i \leftarrow 0$ ($i = 2, \dots, k$); Exit.

(2) [Later entries] $h \leftarrow \min\{i \mid p_i \neq 0\}$; $t \leftarrow p_h$; $p_h \leftarrow 0$; $p_1 \leftarrow t-1$;

$p_{h+1} \leftarrow p_{h+1} + 1$; If $p_k = m$, final exit; Exit .

We utilize the algorithm NEXCOM for the solution procedures in our algorithm CKUC and CKBC.

5.3 Unbounded Change-Making Problem (UCMP)

Consider the formulation CM1 of the UCMP in the previous section. To solve this minimization problem, a good lower bound is needed.

5.3.1 Lower Bounds

It is obvious that for solving UCMP the largest item type would be the first considered for placing in the knapsack. By assuming that the item types satisfy

$$w_1 > w_2 > w_3 > w_j \quad \text{for } j = 4, \dots, n, \quad (5.1)$$

the continuous relaxation of UCMP has an optimal solution as $\bar{x}_1 = c/w_1$, $\bar{x}_i = 0$ for $i = 2, \dots, n$. If $\bar{x}_1$ is not an integer this immediately provides a lower bound for UCMP as $L' = \lfloor c/w_1 \rfloor + 1$. If we consider the continuous solution by imposing an integer value for item type 1, then the continuous optimal solution is:

$$\begin{aligned}\bar{x}_1 &= \lfloor c/w_1 \rfloor, & \bar{x}_2 &= \bar{c}/w_2, \text{ where } \bar{c} \equiv c \pmod{w_1}, \\ \bar{x}_i &= 0, & & \text{for } i = 3, \dots, n.\end{aligned}$$

If $\bar{c} = 0$, then we have the optimal solution, otherwise we have an improved lower bound : $L'' = \lfloor c/w_1 \rfloor + \lfloor \bar{c}/w_2 \rfloor$.

Martello & Toth (1980) have proved that an efficient lower bound for the UCMP can be determined as $L_2 = \min(L^0, L^1)$ where

$$\begin{aligned}L^0 &= \lfloor c/w_1 \rfloor + \lfloor \bar{c}/w_2 \rfloor + \lceil c'/w_3 \rceil, \\ L^1 &= \lfloor c/w_1 \rfloor + \lfloor \bar{c}/w_2 \rfloor - 1 + \lceil (c' + w_1)/w_2 \rceil, \\ \bar{c} &\equiv c \pmod{w_1} \quad \text{and} \quad c' \equiv \bar{c} \pmod{w_2}.\end{aligned}$$

In addition, it is easy to see that if $0 < \bar{c} \leq w_3$, then $L_2 = \bar{x}_1 + 1$. We note that the lower bound may be improved by the following proposition.

Proposition 5.1 Suppose $n > 2$ and $L_2 = \bar{x}_1 + 1$ where the item types satisfy (5.1), $w_s \neq \bar{c}$; $\forall s, 2 \leq s \leq n$ and $\tilde{c} = \bar{c} + w_1 > 2w_2$. Then $L_3 = \bar{x}_1 + 2$ is a lower bound.

Proof Observing that when $\tilde{c} > 2w_2$,

$$L^1 = \lfloor c/w_1 \rfloor + \lfloor \bar{c}/w_2 \rfloor - 1 + \lceil (c' + w_1)/w_2 \rceil \geq \bar{x}_1 + 2.$$

We conclude that $L_2 = L^0$. Further,

$$\bar{x}_1 + 1 = L^0 = \bar{x}_1 + \lfloor \bar{c}/w_2 \rfloor + \lceil c'/w_3 \rceil$$

if and only if $\bar{c} = c' < w_3$. But then there cannot be a feasible solution with $\bar{x}_1 + 1$ items. Hence the lower bound is at least $\bar{x}_1 + 2$, as required. ■

The lower bound for the UCMP may be improved by the following procedure:

LB procedure

Step 1 : Test the condition $\bar{c} < w_3$, if so, then the lower bound for the UCMP is immediately obtained as $\bar{x}_1 + 1$. Otherwise go to Step 2.

Step 2 : Determine the *Martello and Toth bound*.

Step 3 : If current lower bound is $\bar{x}_1 + 1$, then the Proposition 5.1 is used as a criterion to improve the lower bound.

Denote L^* the lower bound determined by the LB procedure.

5.3.2 Algorithm CKUC

Martello & Toth (1990a, p.149) established that the structure of the solution to UCMP's found by MTC1 has 2 properties: (a) the optimal solution value is often equal to the value of bound L_2 , and (b) many equivalent solutions usually exist. We also have empirically observed that: (i) most UCMP's, the lower bounds are $\bar{x}_1 + 1$ or $\bar{x}_1 + 2$, and (ii) the 2 largest types of item are often chosen in the optimal solution.

These properties are very useful in investigating solution procedures. We may initiate a value of each item type based on the lower bound as $x_1 = \bar{x}_1$, $x_j = 0$ for all $j = 2, 3, \dots, n$. If $\bar{c} > 0$, then an optimal may be found by exchanging item type 1 with any other types so as to satisfy the equality constraint. Our algorithm CKUC is based on this idea. The algorithm contains two exchanging procedures:

Procedure 1 (**Pr1**) : one-by-one exchanging,

Procedure 2 (**Pr2**) : composition exchanging.

Pr1 attempts to solve the UCMP such that the objective function value equals L^* by interchanging only one pair of item types. **Pr2** first searches for an optimal solution equal to L^* at each iteration by exchanging a number of items of type 1 with a subset of items of other types generated by NEXCOM.

We may state a corresponding problem to the CMP1 as the following problem CMP2 with an equality of capacity $\hat{c}$ where $\hat{c} = c$ when $L^* = \bar{x}_1 + 1$. If $L^* = \bar{x}_1 + 2$ and $2w_r \neq \bar{c}$ for $r = 2, \dots, n$, then we search for item type $t = \min\{j: w_j < \bar{c}\}$ and set $\hat{c} = c - w_t$.

$$\text{CMP2 :} \quad \text{Solve} \quad \sum_{j=1}^n w_j x_j = \hat{c}, \quad (5.2)$$

$$\sum_{j=1}^n x_j = \bar{x}_1 + \beta, \quad (5.3)$$

$$x_j \geq 0 \text{ and integer}, j = 1, 2, \dots, n. \quad (5.4)$$

Procedure 1 (**Pr1**)

Pr1 is designed to solve the CMP2 with $\beta = 1$, if $L^* = \bar{x}_1 + 1$ or $\beta = 2$, if $L^* = \bar{x}_1 + 2$, by repeatedly inserting and removing a pair of different item types. If both equations (5.2) and (5.3) are satisfied, then the optimal solution to the UCMP is obtained with the optimal value L^* .

Pr1 may be sketched as:

input : $x_j, j = 1, 2, \dots, n$ and r (from Step 2 in the Main algorithm),

output : an optimal solution (if it could be obtained).

Step 1 (initialize) : Let r be the current residual capacity.

Set $x_1 = \bar{x}_1$, $p = \max(j < n : x_j > 0)$ and $q = p + 1$;

Step 2 (remove 1 item type p) : Set $r = r + w_p$; $x_p = x_p - 1$;

Step 3 (search for 2 items such that the total weight sum is equal to c when they are added) :

$rs = r$;

for $i=1$ to n **do**

$rs = rs - w_i$; **if** $rs > w_i$ **then break and go to step 4**;

for $j = 1$ to n **do**

if $w_j = rs$ **then** (an optimal solution is obtained)

$x_p^* = x_p$; $x_i^* = x_i + 1$; $x_j^* = x_j + 1$; **STOP**.

If an optimal solution has not been found, **then** go to step 4.

Step 4 (replace item type q) :

Set $rs = r$; $rr = rs$;

$rr = rr - w_q$; **if** $rr > w_q$ **then** search for t where $t = \min(j; w_j \leq rs)$

if t such that $w_t = rs$ is found **then**

STOP. (an optimal solution is found)

if $t \neq p$ **then** $rs = rs - w_t$; $x_t = x_t + 1$;

else if $t = p = 1$ **go to step 2**;

else if $rr < w_q$ **then** $rs = rr$; $x_q = x_q + 1$;

Step 5 : Repeat step 2- 4 while $x_p > 0$ and an optimal solution has not been found.

Step 6 : Set $p = p+1$ and $q = p+1$;

If $x_p = 0$ and $rr > w_q$ **then** $p = p - 1$; $rs = rs + w_p - w_q$;

$x_p = x_p - 1$;

$x_q = x_q + 1$;

Step 7: While $x_p > 0$ and an optimal solution has not been found, go to the next step in the main algorithm.

The following proposition shows that in case of L^* equal to $\bar{x}_1+1$ or $\bar{x}_1+2$, an optimal solution to the UCMP may be easily obtained by an exchanging method.

Proposition 5.2 Suppose the lower bound L^* for the UCMP equal to $\bar{x}_1+1$ or $\bar{x}_1+2$ and a feasible solution to the problem exists. An optimal solution to the UCMP may be obtained as the solution to the CMP2 by repeatedly exchanging two different item types (p,q) where $p = \max(j < n : x_j > 1)$ and $q = p+1$.

Proof Assume that the item types satisfy (5.1) and the current residual capacity $r > 0$. Denote x_i^* for all $i = 1, 2, \dots, n$ as the optimal solution and suppose $2w_i \neq \bar{c}$ for all i . If $L^* = \bar{x}_1 + 2$ then we set $x_t^* = 1$ where $t = \min(j : w_j < \bar{c})$.

Obviously at step 1, $p = 1, q = 2$. At each iteration (Steps 2 – 4) replace 1 item type p by 1 item type q and then search for an item type h such that w_h is equal to the current residual capacity. By repeated one-by-one exchanges of item type p and q and searching for item type h , clearly, equation (5.3) is always satisfied. If item type h is found, then equation (5.2) is satisfied. Thus a solution to the CM2 is obtained associated with the optimal value L^* that is also the optimal value for the UCMP. ■

If **Pr1** fails to obtain an optimal solution, then **Pr2** will be employed.

Procedure 2 (Pr2)

Consider the following problem (CMP3):

$$\begin{aligned} \text{CMP3 :} \quad & \text{minimize} \quad \beta \\ & \text{subject to} \quad \sum_{j=1}^n w_j x_j = \hat{c}, \\ & \quad \quad \quad \sum_{j=1}^n x_j = \bar{x}_1 + \beta, \\ & \quad \quad \quad x_j \geq 0 \text{ and integer, } j = 1, 2, \dots, n. \end{aligned}$$

It can be seen that the optimal solution to the CMP3 is also the optimal solution to UCMP. **Pr2** solves the CMP3 by increasing the β value. The β value is increased when no solution to both equality constraints (5.2) and (5.3) was found by the previous steps. **Pr2** is based on the idea of exchanging m items of type1 with an m -subset of k item types that is called a composition of m in to k parts. This procedure utilizes NEXCOM and GETFEA. GETFEA is a slight modification of GREEDYUM and it is detailed as follows:

Procedure GETFEA

Let r' be the current residual capacity.

input : n, k, r' and z ;

output : (u_j) , and $\bar{r}$;

GETFEA (n, k, r', z)

Begin

$\bar{r} = r'$;

for $j = k$ to n **do**

if $w_j \leq \bar{r}$ **then do**

$$u_j = \lfloor \bar{r} / w_j \rfloor ;$$

$$\bar{r} = \bar{r} - w_j u_j ;$$

$$z = z + u_j ;$$

End

Pr2 solves the CMP3 by starting with $\beta = 1$. In case the optimal value is not equal to the lower bound, a feasible solution will be searched by GETFEA. The feasible solution will be updated if a better one is found in the next iteration. However, a feasible solution may not be found if the UCMP has no solution. Note that in **Pr2** we use a constant parameter γ to limit the number of parts in a composition. The value of γ may be interpreted as an increasing number of the item types that will be considered in order to exchange with item type 1 at each iteration. The value for γ may be started from 0. Experimentally, we found that an appropriate initial value for γ should be set at

$$\gamma = \begin{cases} 3 & , n < 1000 \\ 2 & , n \geq 1000 . \end{cases}$$

However, γ can be increased in subsequent iterations if an optimal solution has not been found.

Pr2 can be described as :

input : $x_j, j=1, 2, \dots, n$

r (from Step 2 in the main algorithm) and F ,

output : an optimal solution,

Step 1 (initialize) : Set $m = 1$; $x_1 = \bar{x}_1$; $rs = r$;

Step 2 : $rt = rs + mw_1$;

if $rt > (m+1)w_2$ then break ;

else (search for the t^{th} item type where $t = \max(j : \lfloor rt / w_j \rfloor = m)$)

for $i=2$ *to* n *do* *if* $rt/w_i = m+1$ *then* an optimal solution is
 obtained as $x_1^* = x_1 - m$;
 $x_i^* = x_i + m + 1$;
stop ;
else $tt = \lfloor rt / w_i \rfloor$;
if $tt > m$ *then break* ;
else if $tt = m$ *then* $t = i$;

Step 3 : $k = t - 1 + \gamma$;

if $k = n-1$, $m = \bar{x}_1$ *and* $F = 0$, *then stop*. The problem does not have
 an optimal solution.

$v = 1$;

Step 4 : (generate m -subset of k types and determine total weight for each composition)

call NEXCOM (m, k) to generate the v^{th} composition of m into k parts $(p_i)_k$.

$wsum = 0$;

for $i = 1$ *to* k *do* $wsum = wsum + p_i w_i$; $z_i = p_i$;

Step 5: (search for an optimal solution, if it is not found then search for a feasible solution)

$rt = rt - wsum$;

for $ii = 2$ *to* n *do*

if $w_{ii} = rt$ *then* an optimal solution is obtained as

$x_1^* = x_1 - m$; $x_{ii}^* = x_{ii} + 1$;

$x_i = p_i$, for all $i = 2, \dots, k$;

stop .

if $w_{ii} < rt$ *and* $F = 0$ *then do*

$res = rt - w_{ii}$;

for $j = i+1$ *to* n *do* (i in step 4)

if $w_j = res$ **then** a feasible feasible
 solution is obtained as
 $y_1 = y_1 - m$; $y_i = y_i + 1$;
 $y_j = y_j + 1$;
 $y_h = x_h + z_h$, for $h = 2, \dots, k$;
 $F = 1$;
if $w_j < res$ and $F = 0$ **then**
 $res = r'$; $z' = n$;
 call GETFEA($n, j+1, res, \bar{x}_1$)
if GETFEA return $\bar{r} = 0$
then z and (u_j) give a feasible
 solution ;
if $z < z'$ **then** $z' = z$;
 $(u'_j) = (u_j)$;

If an optimal solution is not been found , go to step 6.

Step 6: **If** $p_k \neq m$ **then** $v = v+1$ **and** go to step 4.

If $m < \bar{x}_1$ **then** $m = m + 1$ **and** go to step 2.

If $m = \bar{x}_1$ and the optimal solution cannot be found and $F = 1$ **then** the
 feasible solution is the optimal solution $x_i^* = y_i$ for all $y_i > 0$; $x_i^* = 0$
 otherwise.

Step 7: **If** the optimal solution and a feasible solution are not found **then** $\gamma =$
 $\gamma+1$; Go to step 3.

Pr2 can solve the UCMP to optimality as shown by the following proposition.

Proposition 5.3 Pr2 solves the CMP3 to optimality and an optimal solution to the
 CMP3 is also an optimal solution to the UCMP.

Proof At step 4 of **Pr2**, when m items of type 1 were removed, the algorithm NEXCOM generates each composition of m into $k = t - 1 + \gamma$ parts from item type $j = 2$ to $t + \gamma$ where $t = \max(j : \lfloor rt / w_j \rfloor = m)$ and $\gamma \geq 0$. It is clear that the exchanging maintains the number of items in the knapsack as $\bar{x}_1$.

At step 5, we search for an optimal solution to the CMP3 when $\beta = 1$. If an optimal solution can not be found in the same iteration, then we search for a feasible solution by setting $\beta = 2$. If a feasible solution is not obtained, then the procedure GETFEA is applied to search for a feasible solution with $\beta > 2$. The feasible solution is updated every time a better one is obtained. This implies that β is not greater than the previous iteration. It means that when **Pr2** terminates, an optimal solution to the UCMP is obtained or the problem has no solution if feasible solution is not found. ■

Main algorithm

As mentioned above, the algorithm CKUC mainly contains 2 exchanging procedures. From our empirical analysis, we observed that for the large-sized problems $n \geq 1000$, only the 4 largest item types need to be considered to solve the problem to optimality. In addition, optimal solutions with the optimal values equal to the lower bound are often obtained by **Pr1**. Moreover, it is easy to find an optimal solution by exchanging only a few types next to the item type 1. This is noted by Martello & Toth (1990a, p.153) who remarked that, “*for $n \geq 1000$, the time difference (between MTC1 and MTC2) is considerably higher than the sorting time, indicating that MTC2 takes advantage of the lesser number of item types also in the branch and bound phase*”. They also observed that increasing the data scattering makes the problem more difficult to satisfy the equality constraint. This means that small problems generated in large range tend to hardly obtain an optimal solution. Thus for small problems, complete sorting may be useful for solving the problem efficiently. This leads to the following main algorithm:

Algorithm CKUC

Step 1 (*initallize*) :

- 1.1 Find 4 largest item types $w_1 > w_2 > w_3 > w_4 > w_j$ where $j = 5, \dots, n$ or do complete sorting when $n < 1000$. Compute $\bar{x}_1 = \lfloor c / w_1 \rfloor$ and current residual capacity $rs = c - w_1 \bar{x}_1$. If $rs = 0$ then an optimal solution is obtained as $x_1^* = \bar{x}_1, x_j^* = 0$ for all $j = 2, 3, \dots, n$.
- 1.2 Determine the lower bound L^* by LB procedure. Let $L = \bar{x}_1 + h; h \geq 1$ set $x_1 = \bar{x}_1$ and $x_i = 0$ for all $1 < j \leq n$.

Step 2 (*preprocessing*):

- 2.1 Search for s where $hw_s = \bar{c}$. If s is found, **stop**. An optimal solution is obtained as $x_1^* = \bar{x}_1, x_s^* = h; x_j^* = 0$ for all $j = 2, 3, \dots, n; j \neq s$.
- 2.2 If $h > 1$, call GETFEA($n, 2, c, 0$). If $\bar{r} = 0$ and $z = L^*$ then stop. An optimal solution is obtained as $x_j^* = u_j$ for all $j = 2, 3, \dots, n$.
If $r > 0$, call **Pr1**.
- 2.3 $F = 0$; Call GETFEA($n, 2, rs, \bar{x}_1$). The output may be as follows:
 - 2.3.1 If $\bar{r} = 0$ and $z = L^*$ then an optimal solution is obtained as $x_1^* = \bar{x}_1, x_j^* = u_j$ for all $u_j > 0, x_j^* = 0$ otherwise.
 - 2.3.2 If $\bar{r} = 0$ and $z > L^*$ then save z and (u_j) as a feasible solution; $F = 1$;
 - 2.3.3 If $\bar{r} > 0$ then backtrack to get the value $z = L^* - 1$. Let r be the residual capacity;
Set $x_j = u_j$ for all $u_j > 0$ when $z = L^* - 1$.

Step 3 : Call **Pr1**.

Step 4 : Call **Pr2**.

We illustrate our algorithm CKUC with the following examples.

Example 5.1 (Martello & Toth 1990a, p.140) Let $n = 5$, $c = 29$ and (w_i) are sorted as required by Step 1 (1.1) : (11, 8, 5, 4, 1).

Step 1 (1.1) $\bar{x}_1 = \lfloor 29/11 \rfloor = 2$, $rs = 29 - 2(11) = 7 > 5$,

$$L^* = \min(2+2, 1+3) = 4; \quad (\text{Martello\&Toth lower bound})$$

(1.2) $x_1 = 2$, $x_j = 0$ for all $j = 2, 3, 4, 5$,

$$L^* = \bar{x}_1 + 2, \quad h = 2.$$

Step 2 (2.1) for $j = 2$ to 5, not found s such that $2w_s = 7$.

(2.2) $h = 2$, Call *GETFEA*(5,2,29,0))

for $j = 2$ to 5

$$j = 2; \quad w_2 < 29 \quad \text{then} \quad u_2 = \lfloor 29/8 \rfloor = 3;$$

$$\bar{r} = 29 - 3(8) = 5; \quad z = 0 + 3 = 3.$$

$$j = 3; \quad w_3 = 5 \quad \text{then} \quad u_3 = \lfloor 5/5 \rfloor = 1;$$

$$\bar{r} = 5 - 5 = 0; \quad z = 3 + 1 = 4$$

Therefore an optimal solution is $x_1^* = 0$, $x_2^* = 3$, $x_3^* = 1$, $x_4^* = 0$, $x_5^* = 0$.

Example 5.2 Let $n = 6$, $c = 40$ and the ordered weights $(w_i) = (29, 17, 15, 9, 7, 3)$.

Step 1 (1.1) $\bar{x}_1 = \lfloor 40/29 \rfloor = 1$, $rs = 40 - 29 = 11$, $F = 0$;

$$rs + w_1 = 11 + 29 = 40 > 2(17) \quad \text{and} \quad w_i \neq 11 \quad \text{for} \quad i = 2, 3, 4, 5, 6.$$

$$\text{Thus by proposition 5.1} \quad L^* = \bar{x}_1 + 2 = 1 + 2 = 3.$$

(1.2) $x_1 = 1$, $x_j = 0$ for all $j = 2, 3, 4, 5, 6$.

$$L^* = \bar{x}_1 + 2, \quad h = 2.$$

Step 2 (2.1) for $j = 2$ to 6, not found s such that $2w_s = 11$.

(2.2) $h = 2$, *GETFEA*(6,2,40,0) cannot obtain an optimal solution.

(2.3) Call *GETFEA*(6,2,11,1) for $j = 2$ to 6

$$w_4 < 11 \text{ then } u_4 = \lfloor 11/9 \rfloor = 1 ;$$

$$\bar{r} = 11 - 9 = 2 ; z = 1 + 1 = 2.$$

$$(2.1.3) \quad z = 2 = 3 - 1, r = 2 ,$$

$$x_4 = u_4 = 1.$$

Step 3 An optimal solution cannot be obtained by **Pr1**.

Step 4 Call **Pr2**.

Step 1 : $m = 1 ; x_1 = 1 , rs = 2 .$

Step 2 : $rt = 2 + w_1 = 31 < 2(17)$. $tt = 1$ for $i = 2, 3, 4, 5$, thus $t = 2$.

Step 3 : $k = 2 - 1 + 3 = 4 , v = 1$.

Step 4 : NEXCOM generates 1st composition as $(p_i)_4 = (1, 0, 0, 0) ;$

$$wsum = w_2 = 17.$$

Step 5 : $rt = 31 - 17 = 14 ;$ for $ii = 4 , w_4 = 9 < 14$ and $F = 0 ;$

$$res = 14 - 9 = 5 ; \text{ for } j = 5, 6 , w_j \neq 5 ,$$

and GETFEA fails to get a feasible solution.

Step 6 : $v = 2$ then go to step 4.

Step 4 : The 2nd composition is $(p_i)_4 = (0, 1, 0, 0) ; wsum = w_3 = 15.$

Step 5 : $rt = 31 - 15 = 16 ;$ for $ii = 4 , w_4 = 9 < 16$ and $F = 0 ;$

$res = 16 - 9 = 7 ;$ for $j = 5 , w_5 = 7$ then a feasible solution is obtained

$$\text{as } y_1 = 0 ; y_3 = 1 ; y_4 = 2 ; y_5 = 1 ; F = 1 ;$$

Step 6 : $v = 3$ then go to step 4.

Step 4 : The 3rd composition is $(p_i)_4 = (0, 0, 1, 0) ; wsum = w_4 = 9.$

Step 5 : $rt = 31 - 9 = 22 ;$ for $ii = 4, 5, 6, w_{ii} \neq 22 .$

Step 6 : $v = 4$ then go to step 4.

Step 4 : The 4th composition is $(p_i)_4 = (0, 0, 0, 1) ; wsum = w_5 = 7.$

Step 5 : $rt = 31 - 7 = 24 ;$ for $ii = 4, 5, 6, w_{ii} \neq 24 .$

Therefore the feasible solution is an optimal solution as $x_3^* = 1, x_4^* = 2,$ and $x_5^* = 1,$

$x_j^* = 0$ otherwise.

5.3.3 Computational Results

We implemented our algorithm CKUC in ANSI C on a Silicon Graphic Workstation (SGI R5000) running at clock speed of 150 MHz. To provide a measure of the quality of our algorithm a comparative analysis based on the random set of 4300 instances is carried out. All instances were run on the same machine using an available code that implements the MTC2 algorithm. The integer weights w_i 's are randomly generated from a uniform distribution in $[1, R]$ where $R = 10^3, 10^4, 10^5, 8n, 12n$. We use 11 values of n : 100, 300, 500, 1000, 3000, 5000, 10000, 30000, 50000, 100000, 300000. For each n and R we generated 100 instances. The capacity c is selected as $c = 0.5 \sum_{i=1}^n w_i$.

n	R=10 ³		R=10 ⁴	
	CKUC	MTC2	CKUC	MTC2
100	0.289	0.494	1.647	2.699
300	0.463	1.047	1.833	2.815
500	0.740	1.697	2.075	3.346
1000	0.324	2.119	1.776	4.110
3000	0.496	2.827	2.294	4.867
5000	0.826	3.000	2.386	5.585
10000	1.437	4.523	2.994	6.419
30000	4.121	13.851	5.925	16.418
50000	7.185	23.843	8.944	25.784
100000	15.717	54.332	17.806	56.896
300000	48.632	198.392	50.359	198.573

Table 5.1 : Average CPU time (in milliseconds) of CKUC and MTC2 for Unbounded Change - Making Problem (100 problems) on SGI(R 5000) with capacity $c = 0.5 \sum_{i=1}^n w_i$ with $R = 10^3$ and 10^4 .

n	$R = 10^5$		$R = 8n$		$R=12n$	
	CKUC	MTC2	CKUC	MTC2	CKUC	MTC2
100	26.880	31.631	0.316	0.463	0.402	0.523
300	14.642	24.931	0.701	1.297	0.871	1.533
500	18.952	24.317	1.211	2.205	1.374	2.639
1000	15.095	23.965	1.206	3.469	2.244	4.603
3000	17.446	24.757	4.643	7.181	6.655	11.211
5000	17.221	25.476	8.592	11.932	10.589	16.495
10000	17.701	25.665	17.036	23.111	23.951	31.787
30000	27.377	-	-	-	-	-
500000	32.048	-	-	-	-	-

Table 5.2 : Average CPU time (in milliseconds) of CKUC and MTC2 for Unbounded Change - Making Problem (100 problems) on SGI(R 5000) with capacity $c = 0.5 \sum_{i=1}^n w_i$ with $R = 10^5, 8n$, and $12n$.

The results in table 5.2 may be presented graphically as follows :

Martello & Toth (1990a, pp.151-152) analysed the performance of the algorithm MTC2 only on a number of difficult problems with wide range of data scattering. Table 5.1 gives the average computing times, in milliseconds, of CKUC and MTC2

of for large-sized problem where R is not too large. For all cases CKUC is faster than MTC2 and at least 2 times faster when $n \geq 10^3$. Table 5.2 analyses the performance of the CKUC and MTC2 when $R = 10^5$, $8n$ and $12n$. It confirms the superiority of the CKUC. The computing time of CKUC is slightly faster than MTC2. In the table a “ – ” means that there is not enough memory to run the program on the computer.

From the results, we observe that the Procedure 1 can solve all instances for each value of R as: $R = 10^3$ for all n , $R = 10^4$ and $8n$ when $n \geq 300$, $R = 12n$ when $n \geq 500$ and $R = 10^5$ when $n \geq 30000$. Moreover, when $n \geq R$ most instances can be solved in pre-processing step.

5.4 Bounded Change-Making Problem (BCMP)

The Bounded Change-Making Problem (BCMP) as presented in Section 5.1 may be stated as:

$$\begin{aligned}
 &\text{minimize} && \sum_{j=1}^n x_j \\
 &\text{subject to} && \sum_{j=1}^n w_j x_j = c, \\
 &&& 0 \leq x_j \leq b_j \text{ and integer } , j = 1, 2, \dots, n.
 \end{aligned}$$

The assumptions for solving the problem are as follows:

1. $\sum_{j=1}^n b_j w_j > c$,
2. $b_j w_j \leq c, j = 1, 2, \dots, n$,
3. $w_1 > w_2 > \dots > w_n$.

Martello&Toth (1990a) defined the *break item type* s as: $s = \min \left\{ j : \sum_{i=1}^n b_i w_i > c \right\}$,

$\bar{c} = c - \sum_{j=1}^{s-1} b_j w_j$ and $c' = \bar{c} \pmod{w_s}$. Clearly, the optimal solution of the

continuous relaxed problem is :

$$\bar{x}_i = b_i \text{ for } i = 1, \dots, s-1,$$

$$\bar{x}_s = \lfloor \bar{c} / w_s \rfloor,$$

$$\bar{x}_i = 0 \text{ for } i = s+1, \dots, n.$$

Let $z' = \sum_{j=1}^{s-1} b_j + \lfloor \bar{c} / w_s \rfloor$, thus a lower bound for BCMP can be obtained as $LB' =$

$z' + 1$. They also establish that an efficient lower bound on the optimal value is defined as $LB_2 = \min(LB^0, LB^1)$ where

$$LB^0 = z' + \left\lceil \frac{c'}{w_{s+1}} \right\rceil,$$

$$LB^1 = z' - 1 + \left\lceil \frac{c' + w_{s-1}}{w_s} \right\rceil.$$

Thus, clearly it is easy to see that when $c' \leq w_{s+1}$, the $LB_2 = z' + 1$.

5.4.1 Algorithm CKBC

CKBC is based on the idea of exchanging a subset of item type $j = 1, 2, \dots, s$ with a subset of item type $j = s, s+1, \dots, n$. Due to limiting number of each item types as the upper bound b_j for item type j , the exchanging procedure will remove and insert as many items as allowed by their bounds. We develop **Pr2** of CKUC to optimally solve the BCMP. GETFEA is modified by determining $\hat{u}_j = \min(b_j, \lfloor r' / w_j \rfloor)$ instead of u_j . Again NEXCOM is utilized to generate all number of

available items of each item type j in a subset of $\{1, 2, \dots, s\}$ and in a subset of $\{s, s+1, \dots, n\}$ that can interchange to satisfy the equality constraint. CKBC solves the following problem (BCM1) corresponding to the BCMP:

$$\begin{aligned}
 \text{BCM1 :} \quad & \text{minimize} \quad \beta \\
 & \text{subject to} \quad \sum_{j=1}^n w_j x_j = \hat{c}, \\
 & \quad \sum_{j=1}^n x_j = z' + \beta, \quad (5.4') \\
 & \quad x_j \leq b_j \text{ and integer ; } j = 1, 2, \dots, n. \quad (5.5')
 \end{aligned}$$

The BCM1 is obtained from CM3 when (5.4) and (5.5) are respectively replaced by (5.4') and (5.5').

We modify NEXCOM in procedures 3 and 4 to solve the BCMP. Procedure 4 contains procedure 3. The constant parameters α and λ are used to limit the number of parts (item types) in each composition generated in procedures 3 and 4, respectively.

Procedure 3 (Pr3)

Pr3 generates all compositions of m into α parts for item type $j = s, s+1, \dots, s+\alpha-1$. In addition, the procedure also searches for an optimal solution or a feasible solution for BCM1. This procedure is a modification (step 4 – 6) of CKUC. It can be described as:

Let w_r : sum of the total weight of a composition of m into λ parts for item type $j = s, s-1, \dots, st = s - \lambda + 1$ and the current residual capacity,

zz_j : the j^{th} component of the composition of m into λ parts for item type $j = s, s-1, \dots, s-\lambda+1$.

input : $m, wr, zz_j, st, x_i, w_i, b_i$ for all i ;

output : y_j for all j such that $y_j \neq 0$, if an optimal solution is not obtained.

Step 0 : $v = 1$;

Step 1 : Call NEXCOM(m, α) to generate the v^{th} composition of m into α parts $(q_i)_\alpha$.

(choose only the composition that all numbers of item types satisfy their bounds)

for $i = s$ to α do if $x_i + q_i > b_i$ then go to step 5 .

Step 2 : $wsum = 0$; for $i = s$ to α do $wsum = wsum + q_i w_i$; $z_i = q_i$;

Step 3 : $rs = wr - wsum$; if $rs > wsum$ then return ;

Step 4 : for $i = s$ to n do if $w_i = rs$ and $b_i > x_i$ then an optimal solution is

obtained as $x_i^* = x_i + 1$;

for $j = st$ to $s + \alpha$ do

if $j \leq s$ then $x_j^* = x_j + zz_j$;

if $j \geq s$ then $x_j^* = x_j + q_j$;

stop.

if $w_i < rs$ and $F = 0$ then

$rr = rs - w_i$;

for $j = i+1$ to n do

if $w_j = rr$ and $b_j > x_j$ and $b_i > x_i$ then

for $h = s$ to $s + \alpha$ do

$y_h = x_h$; $y_h = y_h + z_h$;

for $h = 0$ to s do

if $h < s$ then $y_h = x_h$;

if $h > st$ then $y_h = y_h - zz_h$;

$y_j = y_j + 1$; $y_i = y_i + 1$;

$F = 1$;

Step 5 : if $q_{s+\alpha} \neq m$ then $v = v + 1$ and go to step 1 .

Procedure 4 (Pr4)

Pr4 generates each composition of m into λ parts for the item type $j = s, s-1, \dots, st$ where $st = s - \lambda + 1$ then call **Pr3** to solve the BCM1.

input : m, x_i, w_i, b_i for all i ;

Step 1 : $st = s - \lambda + 1 ; k = 1 ;$

Step 2 : Call NEXCOM(m, λ) to generate the k^{th} composition of m into λ parts $(p_i)_\lambda$.

for $i = st$ *to* s *do if* $x_i - p_i < 0$ *then* go to step 6 ;

Step 3 : $ws = 0 ;$ *for* $i = st$ *to* s *do* $ws = ws + p_i w_i ; zz_i = p_i ;$
 $wr = ws + r ;$

Step 4 : *if* $m \geq b_s$ *and* $wr > b_s w_s + (m - b_s + 1) w_{s+1}$ *then* return ;
if $m < b_s$ *and* $wr > (m + 1) w_s$ *then* return ;

Step 5 : Call **Pr3**.

Step 6 : *if* $p_{st} \neq m$ *then* $k = k + 1$, go to step 2 *else* return.

Main algorithm

The algorithm CKBC contains **Pr4** and **Pr3**. For large-sized problems when $n > R$ where R is a range of the random weights, the algorithm needs sorting only a subset of item types. The constant parameter μ is used to limit the number of interchanging. CKBC may be sketched as:

Algorithm CKBC

Step 1 (*initialize*) :

1.1 Sort all item types according to $w_1 \geq w_2 \geq \dots \geq w_n$.

1.2 Compute $sumx = \sum_{j=1}^{s-1} b_j + \lfloor \bar{c} / w_s \rfloor$ and current residual capacity

$$r = c - \sum_{j=1}^{s-1} b_j w_j - \lfloor \bar{c} / w_s \rfloor w_s.$$

If $r = 0$ then an optimal solution is obtained as $x_j^* = b_j, j = 1, 2, \dots, s-1,$

$$x_s^* = \lfloor \bar{c} / w_s \rfloor \text{ and } x_j^* = 0, j = s+1, \dots, n.$$

1.3 Compute the lower bound LB_2 .

Step 2 (*pre-processing*) :

$$rs = r ;$$

Call the modified GETFEA($n, s, rs, sumx$) and the output from the modified GETFEA we consider as 2.3.1 – 2.3.3 in CKUC with replacing u_j by $\hat{u}_j$.

Step 3 : $m = 1$;

Step 4 : Call **Pr4**.

Step 5 : *if* an optimal solution is not found and $m < \mu$ *then* $m = m+1$

go to step 4.

Step 6 : *if* $m = \mu$ *then* backtrack to the updated feasible solution that is an optimal solution.

Experimentally, we found that for $\mu = 9$ all random instances can be solved to optimality by **Pr4** and the optimal value is $LB_2 + 1$. If a feasible solution is not found in the previous steps, the value of μ can be increased in the next step until equal to $sumx$ in step 1. We illustrate our algorithm CKBC with the following example.

Example 5.3 Let $n = 6$, $c = 201$ and the ordered weights $(w_j) = (37, 34, 32, 11, 9, 6)$ with the bound $(b_j) = (3, 4, 3, 4, 1, 1)$.

Step 1 : (1.2) $sumx = 5$, $r = 201 - 3(37) - 2(34) = 22$. $F = 0$;

(1.3) $LB_2 = 6$.

Step 2 : $rs = 22$. Call the modified GETFEA(6,2,22,5) : for $i = 4$, $\hat{u}_4 = \lfloor 22/11 \rfloor = 2$

then a feasible solution is $x_1 = 3$, $x_2 = 2$, $x_4 = 2$, $x_i = 0$ otherwise. $F = 1$;

Step 3 : $m = 1$.

Step 4 : Call **Pr4** :

Step 1 : $st = 2 - 2 + 1 = 1$; $k = 1$; (comment : $\lambda = 2$)

Step 2 : Call NEXCOM(1,2) ; $(p_i)_2 = (1,0)$. $i = 1 : x_1 - 1 = 2 > 0$, $i = 2 : x_2 - 0 = 2 > 0$.

Step 3 : $zz_1 = 1$; $zz_2 = 0$; $ws = 37$, $wr = 37 + 22 = 59$;

Step 4 : $1 < b_2 = 4$ and $59 < 2(34)$;

Step 5 : Call **Pr3**:

step 0 : $v = 1$; **step 1**; $\alpha = 2$, Call NEXCOM(1,2); $(q_i)_2 = (1,0)$

step 2 : $wsum = q_2 w_2 = 34$; **step 3** : $rs = 59 - 34 = 25$. **step 5** : $v = 2$.

step 1: $\alpha = 2$, Call NEXCOM(1,2); $(q_i)_2 = (0,1)$

step 2 : $wsum = q_3 w_3 = 32$; **step 3** : $rs = 59 - 32 = 27$.

Step 6 : $k = 2$ go to step 2 .

Step 2 : Call NEXCOM(1,2) ; $(p_i)_2 = (0,1)$. $i = 1 : x_1 - 0 = 3 > 0$, $i = 2 : x_2 - 1 = 1 > 0$.

Step 3 : $zz_1 = 0$; $zz_2 = 1$; $ws = 32$, $wr = 32 + 22 = 54$;

Step 4 : $1 < b_2 = 4$ and $54 < 2(34)$;

Step 5 : Call **Pr3**:

step 0 : $v = 1$;

step 1; $\alpha = 2$, Call NEXCOM(1,2); $(q_i)_2 = (1,0)$;

step 2 : $wsum = q_2 w_2 = 34$; **step 3** : $rs = 54 - 34 = 20$. **step 5** : $v = 2$.

step 1: $\alpha = 2$, Call NEXCOM(1,2); $(q_i)_2 = (0,1)$;

step 2 : $wsum = q_3 w_3 = 32$; **step 3** : $rs = 54 - 32 = 22$.

Step 6 : $p_2 = m = 1$ *return* .

Step 6 : $m = 2$.

Step 4 : Call **Pr4** :

Step 2 : Call NEXCOM(2,2) ; $(p_i)_2 = (2,0)$. $i = 1$, $x_1 - 2 = 1 > 0$; $i = 2$, $x_2 - 0 = 2 > 0$.

Step 3 : $zz_1 = 2$; $zz_2 = 0$; $ws = 2(37) = 74$; $wr = 74 + 22 = 96$;

Step 4 : $2 < b_2 = 4$ and $96 < 3(34)$;

Step 5 : Call **Pr3**:

step 0 : $v = 1$;

step 1 : $\alpha = 2$, Call NEXCOM(2,2); $(q_i)_2 = (2,0)$

step 2 : $wsum = q_2 w_2 = 2(34) = 68$; **step 3 :** $rs = 96 - 68 = 28$.

step 5 : $v = 2$.

step 1 : $\alpha = 2$, Call NEXCOM(2,2); $(q_i)_2 = (0,2)$

step 2 : $wsum = q_3 w_3 = 2(32) = 64$; **step 3 :** $rs = 96 - 64 = 32$.

step 4 : $i = 3$; $w_3 = 32 = rs$ and $b_3 = 3 > 0$ then $x_3^* = 0 + 1 = 1$, *an optimal solution is obtained as :*

$x_1^* = 3 - 2 = 1$; $x_2^* = 2 - 0 = 2$; $x_3^* = 1 + 2 = 3$; $x_j^* = 0$; otherwise.

5.4.2 Computational Results

We also implemented our algorithm CKBC in ANSI C on a Silicon Graphic Workstation (SGI R5000) running at clock speed of 150 MHz. To provide a measure of the quality of our algorithm a comparative analysis based on the random set of 3800 instances is carried out. All instances were run on the available code that implements MTCB on the same machine. The integer weights w_i 's are randomly generated from a uniform distribution in $[1, R]$ where $R = 10^3, 10^4, 10^5$,

8n. We use 11 values of n : 100, 300, 500, 1000, 3000, 5000, 10000, 30000, 50000, 100000, 300000. The bound b_i is randomly distributed in $[1,20]$ and the capacity c is selected as $c = 0.5 \sum_{i=1}^n w_i$. For each n and R we generated 100 instances.

n	$R=10^3$		$R=10^4$	
	CKBC	MTCB	CKBC	MTCB
100	0.353	0.552	1.734	3.330
300	0.693	1.122	2.018	3.532
500	1.004	1.755	2.751	4.215
1000	2.061	3.400	3.404	6.122
3000	3.197	10.920	7.528	13.738
5000	5.847	19.325	12.511	22.266
10000	13.822	44.075	25.157	48.191
30000	50.808	166.781	57.769	170.832
50000	75.866	310.374	104.703	311.417
100000	170.747	754.970	329.063	780.886
300000	688.238	3095.630	857.039	3078.050

Table 5.3 : Average CPU time (in milliseconds) of CKBC and MTCB for Bounded Change - Making Problem (100 problems) on SGI(R 5000) with capacity $c = 0.5 \sum_{i=1}^n w_i$, $R = 10^3, 10^4$ and bound $b_i \in [1,20]$.

The results in table 5.3 may be presented graphically as follows:

n	$R = 10^5$		$R = 8n$	
	CKBC	MTCB	CKBC	MTCB
100	22.624	41.792	0.314	0.508
300	13.834	30.710	0.929	1.471
500	15.324	29.653	1.532	2.527
1000	18.637	31.152	3.220	5.690
3000	21.718	40.375	13.768	18.584
5000	29.899	52.396	18.264	32.093
10000	40.514	76.366	37.694	69.956
30000	102.412	202.503	146.504	-

Table 5.4 : Average CPU time (in milliseconds) of CKBC and MTCB for Bounded Change - Making Problem (100 problems) on SGI(R 5000) with capacity $c = 0.5 \sum_{i=1}^n w_i$, $R = 10^5, 8n$ and bound $b_i \in [1,20]$.

The results in table 5.4 may be presented graphically as follows:

Table 5.3 compares the average computing times (in milliseconds) of CKBC with MTCB for all values of n when $R = 10^3$ and 10^4 . The table shows that for all cases CKBC solves the problems faster than MTCB. Moreover, when $n > R$, CKBC is much faster than MTCB. We observed that for large-sized problems an optimal solution is obtained by exchanging a few item types around the break item. Therefore, when $n > R$, CKBC requires sorting a subset of item types which contains only the 3% largest item types. Thus in CKBC, we modify the quick sort algorithm to sort only a subset of item types. Table 5.4 gives the average computing times of the both algorithms for the more difficult problems since the weights are randomly in $R = 10^5$ and $8n$. Due to computer memory, CKBC can solve problems only up to $n = 30000$, while MTCB can solve problems only n up to 10000, when $R = 8n$. For all cases, the superiority of CKBC is established in the both tables.

5.5 Conclusions

This chapter presents two exact algorithms CKUC and CKBC for the Unbounded Change-Making Problem and Bounded Change-Making Problem. Our algorithms may be viewed as utilizing a branch and search technique. The exchanging procedures are the techniques for branching by decreasing the number of the items that are first filled in the knapsack and increasing the number of items that can make the knapsack full. Both algorithms perform faster than MTC2 and MTCB since we exploit the branching technique by utilizing the algorithm NEXCOM. Moreover a feasible solution is also searched to be updated and stored when a better one is found during the branching phase by the procedure GETFEA. This technique may save the computing time when the optimal function value is not equal to lower bound.

By comparing the computing times in cases of the same value of problem size and range of random data value, we also observe as Martello and Toth (1990a, p.156) remark that “*The solution of BCMP appears harder than that of UCMP.*” For our algorithms CKUC and CKBC, the reason is that in CKUC, the algorithm NEXCOM is used to generate the compositions for only one set of item types but in CKBC, it is used to generate the compositions for two sets. In addition, for sorting time, a partial sorting (3% of the largest item types) is needed in solving BCMP, while only the four largest item types are required for any size of UCMP.

CHAPTER 6

Conclusions

The Knapsack Problem (KP) is one of the important integer linear programming problems that has been intensively studied over the past forty years or so. Practical and theoretical reasons have motivated interest in this problem. On the practical side this NP-hard problem arises in many applications such as financial management, cargo loading, cutting stock and cryptography etc. On the theoretical side, the KP arises as a subproblem in the solution of many integer programming problems. In addition a general integer programming problem can be transformed in to a KP. The KP is typically concerned with selecting from a set of n given items, each with a specified weight and value, a subset of items whose weight sum does not exceed a prescribed capacity and whose value is maximum. Numerous variations of Knapsack

Problem have been studied such as 0-1Knapsack Problem (0-1KP), Bounded Knapsack Problem (BKP), Unbounded Knapsack Problem (UKP), and Change-Making Problem (CMP), etc.

Over the years researchers have proposed numerous exact algorithms based on either Dynamic Programming or Branch and Bound. Basically all these algorithms require sorting the profit-to-weight ratios of items. In fact, much of the computational time required for solving Knapsack Problems is attributed to sorting. Consequently, in the last two decades, considerable effort has gone into developing algorithms which sort only a chosen small subset of items the so-called “core”. This has led to the solution of large-sized Knapsack Problems. The better performing literature algorithms include:

- COMBO for the 0-1KP (Martello, Pisinger & Toth, 1997).
- BOUKNAP for the BKP (Pisinger, 1995b).
- MTU2 for the UKP (Martello & Toth, 1990a).
- MTC2 for the Unbounded CMP (Martelo & Toth, 1990a).
- MTCB for the Bounded CMP (Martello & Toth, 1990a).

Computational results indicate that the higher the correlation between profit and weight, the more difficult the problem. This has given rise to classes of difficult Knapsack Problems, in particular: strongly correlated data (profit is linearly related to the weight of an item) and subset sum data (profit and weight are equal) types. On these data types, the above mentioned algorithms do not perform well with the exception of COMBO. COMBO performs well on all data types, however it's CPU time for the strongly corrected data and subset sum data types is larger compared to other data types.

This thesis has focused on the hard Knapsack Problems: 0-1KP, BKP, UKP and CMP. We have developed, presented and extensively tested 9 new specialized algorithms as detailed below:

- Algorithms CK1, CK2, CK3 for the 0-1KP with subset sum data, strongly correlated data and inverse strongly correlated data respectively. These are compared with the available literature coded algorithm COMBO. Our new algorithms performed very well for all large-sized problems (more than 1000 items). Indeed, we have demonstrated that problems with up to one million items can be solved efficiently.
- Algorithms CK4 and CK5 for the BKP with subset sum data and strongly correlated data, respectively. These are compared with the available literature coded algorithm BOUKNAP. CK4 and CK5 can solve problems with up to one million items while BOUKNAP could only solve problems with up to 300000 items.
- Algorithms CKU1 and CKU2 for the Unbounded Strongly Correlated Knapsack Problem (UNSCK) and Unbounded Subset Sum Problem (UNSSP), respectively. Both algorithms are compared with the available literature coded algorithm MTU2. CKU1 and CKU2 performed better in most instances than MTU2 and all test problems with up to one million items are solved to optimality.
- Algorithms CKUC for the Unbounded Change-Making Problem and CKBC for the Bounded Change-Making Problem are compared with the available codes algorithm MTC2 and MTCB, respectively. Both of our algorithms perform faster than MTC2 and MTCB since we exploit the branching technique by utilizing the algorithm NEXCOM.

Our major contribution is the developed of new algorithms that are not based on dynamic programming or branch and bound techniques. We have demonstrated that our algorithms can solve recognized difficult classes of problems. On the basis of extensive computational results our algorithms perform well against the codes available in the literature.

Whilst our computational work has been extensive there are many issues that could be addressed. In particular, the following are worthwhile computational projects for further research.

- Test the algorithms performance by ranging the knapsack capacity c . That is consider $c = a \sum_{i=1}^n w_i$ with $0 < a < 1$. We have mostly used $a = 0.5$ and $a = 0.1$.
- Use of different data sets. We have generated all our data randomly using a uniform distribution. For large n this provides a wide range of item weights and thus the problems could be viewed as easier. Indeed, in almost all cases our solutions fully utilize the knapsack capacity. A full enumeration could be required to identify optimal solutions which do not fully fill the knapsack. It would be interesting to test data sets that are normally distributed.
- Code the literature algorithms so that a more detailed analysis of performance could be made. This would allow the opportunity to experiment with different computer architectures and compilers.

References

Andonov, R., Poirriez, V., and Rajopadhye, S. 1997, 'Unbounded Knapsack Problem : Dynamic Programming Revisited', Publication interne n °PI-1152 Institut De Recherche En Informatique Et Systemes Aleatoires (October 1997).

Aherns J.H. and Finke G. 1975, 'Merging and Sorting Applied to the 0-1 Knapsack Problem', *Operations Research*, vol.23, pp.1099-1109.

D'Atri G. and Puech C. 1982, 'Probabilistic Analysis of the Subset-Sum Problem', *Discrete Applied Mathematics*, vol. 4, pp. 329-334.

Balas, E. 1965, 'An Additive Algorithm for Solving Linear Programs with Zero-One Variables', *Operations Research*, vol. 13, no. 4, pp. 517-546.

Balas, E. 1975, 'Facet of the Knapsack Polytope', *Mathematical Programming*, vol. 8, pp. 146-164.

Balas, E. and Zemel, E. 1978, 'Facets of the Knapsack Polytope from Minimal Covers', *SIAM Journal of Applied Mathematics*, vol. 34, pp. 119-148.

Balas, E. and Zemel, E. 1980, 'An Algorithm for Large Zero-One Knapsack Problem', *Operations Research*, vol. 28, pp. 1130-1154.

Balintfy, J.L., Ross, G.T., Sinha, P. and Zoltners, A.A. 1978, 'A mathematical programming system for preference and compatibility maximized menu planning and scheduling', *Mathematical Programming*, vol. 15, pp. 63-76.

Bellman, R. 1954, 'Some applications of the theory of dynamic programming—a review', *Operations Research*, vol. 2, pp.275-288.

Bellman R. 1957, *Dynamic Programming*, Princeton NJ.

Bellman R. and Dreyfus, S.E. 1962, *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ.

Bjorndal, M.H., Capara, A., Cowling, P.I., Della Croce, F., Lourenco, H., Malucelli, F., Orman, A.J., Pisinger, D., Rego, C., and Salazar, J.J. 1995, 'Some Thoughts on Combinatorial Optimisation', *European Journal of Operational Research*, vol. 83, pp. 253-270.

Brown, J. R. 1979, 'The Knapsack Sharing Problem', *Operations Research*, 1979, vol. 27, pp. 341-355.

Burkard, R. E. and Pferschy, U. 1994, 'The Inverse-parametric Knapsack Problem', Technische Universität Graz, Austria, Report 94/288.

Cabot, A.V. 1970, 'An Enumeration Algorithm for Knapsack Problems', *Operations Research*, vol. 18, pp.306-311.

Caccetta, L. and Kulanoor, A. 1998, 'Algorithms for Some Hard Knapsack Problems', Submitted.

Capara, A., Kellerer, H., Pferschy, U., and Pisinger, D. 1998, 'Approximation Algorithms for Knapsack Problems with Cardinality Constraints', *Technical Report, DIKU, University of Copenhagen 98/4*.

Chajakis, E. and Guignard, M. 1994, 'Exact Algorithms for the Setup Knapsack Problem', *INFOR*, vol. 32, no. 3, pp. 124-142.

Chang, S.K. and Gill, A. 1970, 'Algorithmic Solution of the Change-Making Problem', *Journal of ACM*, vol. 17, pp. 113-122.

Chang, S.K. and Korsh, J.F. 1976, 'Canonical Coin-Changing and Greedy Solutions', *Journal of ACM*, vol. 23, pp. 418-422.

Chvatal V. 1980, 'Hard Knapsack Problem', *Operations Research*, vol. 28, pp. 402-411.

Cord, J. 1964, 'A Method for Allocating Funds to Investment Projects when Returns Are Subject to Uncertainty', *Management Science*, vol. 10, no. 2, pp. 335-341.

Crowder, H., Johnson, E., and Padberg, M. 1983, 'Solving Large-Scale Zero-One Linear Programming Problems', *Operations Research*, vol. 31, no. 4, pp. 803-833.

Dantzig G.B. 1957, 'Discrete Variables Extremum Problems', *Operations Research*, vol. 5, pp. 266-277.

Dembo, R.S. and Hammer, P.L. 1980, 'A Reduction Algorithm for Knapsack Problems', *Methods of Operations Research*, vol. 36, pp. 49-60.

Diffie, W. and Hellman, M.E. 1976, 'New directions in cryptography', *IEEE Transactions Information Theory*, IT-36, pp.644-654.

Driebeek, N. J. 1966, 'An Algorithm for the Solution of Mixed Integer Programming Problems', *Management Science*, vol. 12, pp. 576-587.

Dudzinski, K. 1991, 'A Note on Dominance Relation in Unbounded Knapsack Problems', *Operations Research Letters*, vol. 10, pp.417-419.

Dudzinski, K. and Walukiewicz, S. 1987, 'Exact Methods for the Knapsack Problem and its Generalizations', *European journal of Operational Research*, vol. 28, pp. 3-21.

Elmaghraby, S., and Wig, M. 1970, ' On the Treatment of Stock Cutting Problems as Diophantine Programs', Operations research Center Report no. 61, North Carolina State University at Raleigh, May 11, 1970.

Faaland B. 1973, 'Solution of the Value-independent Knapsack Problem by Partitioning', *Operations Research*, vol. 21, pp.332-337.

Fayard, D. and Plateau, G. 1982, 'An Algorithm for the Solution of the 0-1 Knapsack Problem', *Computing*, vol. 28, no. 3, pp. 269-287.

Fayard, D. and Plateau, G. 1994, 'An Exact Algorithm for the 0-1 Collapsing Knapsack Problem', *Operations Research Letters*, vol. 7, pp. 181-187.

Fischetti M. 1990, 'A New Linear Storage, Polynomial-time Approximation Scheme for the Subset-Sum Problem', *Discrete Applied Mathematics*, vol. 26, pp. 61-77.

Garfinkel, R.S. and Nemhauser, G.L. 1972, *Integer Programming*, John Wiley and Sons, New York.

Gavish, B. and Pirkul, H. 1985, 'Efficient Algorithms for Solving Multiconstraint Zero-One Knapsack Problems to Optimality', *Mathematical Programming*, vol. 31, pp. 78-105.

Geoffrion, A.M. 1974, 'Lagrangian Relaxation for Integer Programming', *Mathematical Programming Study*, vol. 2, pp. 82-114.

Gilmore P.C and Gomory R.E. 1963, 'A Linear Programming Approach to the Cutting Stock Problem II', *Operations Research*, vol. 11, pp.863-888.

Gilmore P.C. and Gomory R.E. 1966, 'The Theory and Computation of Knapsack Functions', *Operations Research*, vol.14, pp.1045-1074.

Glover, F. 1972, 'New Results for Reducing Linear Programs to Knapsack Problems', *Management Science Report Series 72-7*, The University of Colorado at Boulder.

Gottlieb, E. S. 1990, 'Facet Defining Inequalities for the Dichotomous Knapsack Problem', *Operations Research Letters*, vol. 9, pp. 21-29.

Greenberg, H. 1986, 'On Equivalent Knapsack Problems', *Discrete Applied Mathematics*, vol. 14, pp.263-268.

Greenberg H. and Hegerich R.L. 1970, 'A Branch Search Algorithm for the Knapsack Problem', *Management Science*, vol. 16, pp. 327-332.

Hansmann, F. 1961, 'Operations Research in the National Planning of Underdeveloped Countries', *Operations Research*, vol. 9, pp. 203-248.

Hammer, P. L., Johnson, E. L. and Peled, U. N. 1975, 'Facet of Regular 0-1 Polytopes', *Mathematical Programming*, vol. 8, pp. 179-206.

Holmberg, K. and Jörnsten, K. 1996, 'Solving the Generalized Knapsack Problem with Variable Coefficients', *Naval Research Logistics*, vol. 43, pp. 673-689.

Horowitz, E. and Sahni, S. 1974, 'Computing Partitions to the Knapsack Problem', *Journal of the Association for Computing Machinery*, vol. 21, no. 2, pp. 277-292.

Hu, T. C. 1969, '*Integer Programming and Network Flows*', Addison-Wesley, New York.

Hu, T.C. and Lenard, M.L. 1976, 'Optimality of a Heuristic Solution for a Class of Knapsack Problems', *Operations Research*, vol. 24, pp.193-1996.

Ibarra O.H. and Kim C.E. 1975, 'Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems', *Journal of ACM*, vol. 22, pp. 463-468.

Ingargiola G.P. and Korsh J.F. 1973, 'A Reduction Algorithm for Zero-One Single Knapsack Problems', *Management Science*, vol. 20, pp. 460-463.

Janakiram V.K., Gehringer E.F., Agrawal D.P., and Mehrotra R. 1988, 'A Randomized Parallel Branch-and-Bound Algorithm', *International Journal of Parallel Programming*, vol. 17, pp.277-300.

- Johnson D.S. 1974, 'Approximation Algorithms for Combinatorial Problems', *Journal of Computer and System Science*, vol.9, pp. 256-278.
- Kadane, J. B. 1968, 'Discrete Search and Neyman-Pearson Lemma', *Journal of Mathematical Analysis and Applications*, vol. 22, no.1, p.156.
- Kaplan, S. 1966, 'Solution of the Lorie-Savage and Similar Integer Programming Problems by the Generalized Lagrange Multiplier Method', *Operations Research*, vol. 14, no. 6, pp.1130-1136.
- Kendell, K. and Zionts, S. 1972, 'Solving Integer Programming Problems by Aggregating Constraints', School of Management, *Working Paper no. 155*, S.U.N.Y. at Buffalo.
- Kianfar, F. 1971, 'Stronger Inequalities for 0,1 Integer Programming Using Knapsack Functions', *Operations Research*, vol. 19, pp. 1374-1392.
- Kindervater G.A.P. and Trienekens H.W.J.M. 1988, 'Experiments with Parallel Algorithms for Combinatorial Problems', *European Journal of Operational research*, vol. 33, pp.65-81.
- Kleywegt, A. J. and Papastavrou, J. D. 1998, 'The Dynamic and Stochastic Knapsack Problem', *Operations Research*, vol. 40, no. 1, pp.17-35.
- Kolesar P.J. 1967, 'A Branch and Bound Algorithm for the Knapsack Problem', *Management Science*, vol.13, pp. 723-735.
- Lai T.C. and Sahni S. 1984, 'Anomalies in Parallel Branch-and-Bound Algorithms', *Communication of ACM*, vol. 27, pp. 594-602.

Lagarias J.C. and Odlyzko A.M., 1983, 'Solving Low-density Subset Sum Problem', *Proceeding 24th Annual IEEE Symposium Foundations of Computer Science* , pp.1-10.

Lawler E.L. 1979, 'Fast Approximation Algorithms for Knapsack Problems', *Mathematics of Operations Research* , vol. 4, pp. 339-356.

Lee J., Shragowitz E. and Sahni S. 1988, 'A Hypercube Algorithm for the 0/1 Knapsack Problem', *Parallel and Distributed Computing*, vol. 5, pp. 438-456.

Lin J. and Storer J.A. 1991, 'Processor-efficient Hypercube Algorithms for the Knapsack Problem', *Journal of Parallel and Distributed Computing*, vol.11, pp.332-337.

Loot W. and Smith T.H.C. 1992, 'A Parallel Algorithm for the 0-1 Knapsack Problem', *International Journal of Parallel Programming*, vol. 21, pp. 349-362.

Lueker, G.S. 1975, 'Two NP-complete Problems in Nonnegative Integer Programming. Report No. 178' *Computer Science Laboratory*, Princeton University, Princeton, NJ.

Magazine, M.J., Nemhauser, G.L. and Trotter, JR. L.E. 1975, 'When the Greedy Solution Solves a Class of Knapsack Problems', *Operations Research*, vol. 23, pp. 207-217.

Magazine, M.J. and Oguz, O. 1981, 'A fully Polynomial approximate algorithm for 0-1 Knapsack Problem', *European Journal of Operational Research*, vol. 8, pp. 270-273.

Majchrzak, J. 1980, 'On Relations Between Continuous and Discrete Multicriteria Optimization Problems, *Proceedings of 9th IFIP Conference on Optimization Techniques*, Lecture Notes in Control and Optimization Sciences 28, Springer-Verlag, Berlin, pp. 473-481.

Martello S., Pisinger D., and Toth P. 1997, 'Dynamic Programming and Tight Bounds for the 0-1 Knapsack Problem, working paper DIKU, University of Copenhagen, July 29, 1997.

Martello, S. and Toth, P. 1977, 'An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound Algorithm', *European Journal of Operational Research*, vol. 1, pp.169-175.

Martello, S. and Toth, P. 1980, 'Optimal and Canonical Solutions of the Change-Making Problem', *European Journal of Operational Research*, vol. 4, pp.322-329.

Martello S. and Toth P. 1984, 'A Mixture of Dynamic Programming and Branch-and-Bound for the Subset-Sum Problem, *Management Science*, vol. 30, pp. 765-771.

Martello S. and Toth P. 1987, 'Algorithms for Knapsack Problems', in *Surveys in Combinatorial Optimization*, eds. Martello, S., Laporte, G., Minoux, M., and Ribeiro, C., Elsevier science publishers B.V., North-Holland Amsterdam Netherlands.

Martello, S. and Toth, P. 1988, 'A New Algorithm for the 0-1 Knapsack Problem', *Management Science*, vol. 34, pp. 633-644.

Martello, S. and Toth, P. 1990a, '*Knapsack Problems: Algorithms and Computer Implementations*', John Wiley and Sons.

Martello S. and Toth, P. 1990b, 'An Exact Algorithm for Large Unbounded Knapsack Problems', *Operations Research Letters*, vol. 9, pp.15-20.

Martello, S. and Toth, P. 1997, 'Upper Bounds and Algorithms for Hard 0-1 Knapsack Problems', *Operations Research*, vol. 45, no. 5, pp.768-778.

Morin, T.L. and Marsten, R.E. 1976, 'An Algorithm for Nonlinear Knapsack Problems', *Management Science*, vol. 22, pp. 1147-1158.

Müller – Merbach , H. 1978, 'An Improved Upper Bound for the Zero-One Knapsack Problem: A note on the paper by Martello and Toth', *European Journal of Operational Research*, vol. 2, pp. 212-213.

Nauss R.M. 1976, 'An Efficient Algorithm for the 0-1 Knapsack Problem', *Management Science*, vol. 23, pp. 27-31.

Nemhauser, G. L. and Wolsey, L. A. 1988, '*Integer and Combinatorial Optimization*', John Wiley and Sons.

Nijenhuis, A. and Wilf, H.S. 1975, '*Combinatorial Algorithms*', Academic Press, Inc.USA.

Pandit S.N.N. and Ravi Kumar M. 1993, 'A Lexicographic Search for Strongly Correlated 0-1 Knapsack Problems, *Opsearch*, vol. 30, pp. 97-116.

Pferchy U. 1998, 'Stochastic Analysis of Greedy Algorithms for the Subset Sum Problem', working paper, Universitat Graz, Institut für Statistik und Operations Research, Austria.

Pisinger D. 1995a, 'An Expanding-core Algorithm for the Exact 0-1 Knapsack Problem', *European Journal of Operational Research*, vol. 87, pp. 175-187.

Pisinger D. 1995b, *Algorithms for Knapsack Problem*, Ph.D. thesis.

Pisinger D. 1997, 'A Minimal Algorithm for the 0-1 Knapsack Problem', *Operations Research*, vol. 45, pp. 758-767.

Pisinger D. 1998, 'A Fast Algorithm for Strongly Correlated Knapsack Problems', *Discrete Applied Mathematics*, vol. 89, pp. 197-212.

Pisinger, D. and Toth, P. 1998, 'Knapsack Problems' in *Handbook of Combinatorial Optimization*, vol. 1, eds. Du, D.-Z. and Pardalos, Kluwer Academic Publishers, pp. 299-428.

Polak E. and Payne, A. N. 1976, 'On Multicriteria Optimization' in *Directions in Large-Scale Systems*, eds. Ho, Y. C. and Mitter, S. K., Plenum Press, New York pp. 77-94.

Sahni S. 1975, 'Approximate Algorithms for the 0-1 Knapsack Problem', *Journal of ACM*, vol. 22, pp. 115-124.

Salkin, H. M. 1975, *Integer Programming*, Addison-Wesley Publishing Company, Inc.

Salkin, H. M. and de Kluyver, C. A. 1975, 'The Knapsack Problem : a survey', *Naval Research Logistics Quarterly*, vol. 22, pp. 127-144.

Shapiro, J. F. 1968, 'Dynamic Programming Algorithms for the Integer Programming Problem-I: The Integer Programming Problem Viewed as a Knapsack Type Problem', *Operations Research*, vol. 16, no. 1, pp. 103-121.

Shapiro, J. F. and Wagner, H. M. 1967, 'A Finite Renewal Algorithm for the Knapsack and Turnpike Models', *Operations Research*, vol. 15, no. 2, pp. 319-341.

Shih, W. 1979, 'A Branch and Bound Method for the Multiconstraint Zero-One Knapsack Problem', *Journal of the Operational Research Society*, vol. 30, pp. 369-378.

Soma N.Y. and Zinober A.S.I., Yanasse H.H. and P.J.Harley P.J. 1995, 'A Polynomial Approximation Scheme for the Subset Sum Problem', *Discrete Applied Mathematics*, vol. 57, pp. 243-253.

Suzuki, H. 1978, 'A Generalized Knapsack Problem with Variable Coefficients', *Mathematical Programming*, vol. 15, pp. 162-176.

Syslo, M., Deo, N. and Kowalik, J. S. 1983, *Discrete Optimization Algorithms with PASCAL programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Tien, B.N., and Hu, T.C. 1977, 'Error Bounds and the Applicability of the Greedy Solution to the Coin-Changing Problem ', *Operations Research*, vol. 25, pp. 404-418.

- Tomlin, J. A. 1970, 'Branch and Bound Methods for Integer and Non-Convex Programming', in *Integer and Nonlinear Programming*, ed. Abadie, J. North-Holland, Amsterdam.
- Tomlin, J. A. 1971, 'An Improved Branch and Bound Method for Integer Programming', *Operations Research*, vol. 19, pp. 1070-1075.
- Toth P. 1980, 'Dynamic Programming Algorithms for the Zero-One Knapsack Problem', *Computing*, vol. 25, pp. 29-45.
- Van Roy, T. J. and Wolsey, L. A. 1987, 'Solving Mixed Integer Programming Problems Using Automatic Reformulation', *Operations Research*, vol. 35, no. 1, pp. 45-57.
- Weismantel, R. 1997, 'On the 0/1 Knapsack Polytope', *Mathematical Programming*, vol. 77, no. 1, pp.49-68.
- Wolsey, L. A. 1975, 'Faces of Linear Inequalities in 0-1 Variables', *Mathematical Programming*, vol. 8, pp. 165-178.
- Wright, J.W. 1975, 'The Change-Making Problem', *Journal of the Association for Computing Machinery*, vol. 22, pp. 125-128.
- Yamada, T. and Futakawa, M. 1997, 'Heuristic and Reduction Algorithms for the Knapsack Sharing Problem', *Computers and Operations Research*, vol. 24, no.10, pp. 961-967.
- Yu, G. 1996, 'On the Max-Min 0-1 Knapsack Problem with Robust Optimization Applications', *Operations Research*, vol. 44, no. 2, pp. 407-415.

Zoltners, A. A. 1978, 'A Direct Descent Binary Knapsack Algorithm', *Journal of ACM*, vol. 25, pp. 304-311.