A Complementary Heuristic for the Unbounded Knapsack Problem

Swarna Chitra Iyer

A thesis submitted in fulfilment of the requirements for the Masters' degree

Department of Computer and Mathematical Sciences
Victoria University of Technology

December, 1997

Declaration

I hereby certify that:

1. the following thesis contains only my original work,
2. due acknowledgment has been made in the text of the thesis to all other material used and
3. the thesis is less than 60,000 words in length, exclusive of tables, figures and footnotes.

Swarna Chitra Iyer
December, 1997

ABSTRACT

As a solution algorithm for Unbounded Knapsack Problem, the performance analysis of density-ordered greedy heuristic, weight-ordered greedy heuristic, value-ordered greedy heuristic, extended greedy heuristic and total-value heuristic has been done. Empirical experiments on different test problems have been analysed and reported. Problem instances with a very large number of undominated items were generated in addition to the types of instances suggested by Martello and Toth (1990). Theoretically, the lower bound on the performance for total-value heuristic is better than the corresponding lower bounds for the density-ordered greedy heuristic and the extended greedy heuristic as discussed by White (1992) and Kohli and Krishnamurti (1992). The computational tests fail to show clear superiority of any particular heuristic algorithm, although each heuristic produces good quality solutions. If the combination of the density-ordered greedy and the total-value greedy heuristics are considered then the combination shows complementary effect. A new heuristic algorithm incorporating the structural properties of the density-ordered greedy heuristic and the total-value greedy heuristic is developed and its complementary effect studied. It was found that the combination of the density-ordered greedy heuristic, the extended greedy heuristic, the total-value greedy heuristic and the new complementary heuristic gives a better performance result than the single best heuristic in the combination.

Acknowledgments

I would like to thank:

- my parents, Sharada and Subramanian
- my supervisor, Lutfar Khan
- my colleagues, particularly Mehmet Tat, Tony Sahama, and Damon Burgess and P. Rajendran (Raj) for their technical support,

and a special thanks to my husband Soundarajan.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

1. FUNDAMENTALS

Knapsack problems are intensively studied mainly for their simple structure which, on the one hand allows exploitation of a number of combinatorial properties and, on the other, facilitates the solution of more complex optimisation problems through a series of knapsack-type subproblems.

In the following sections we shall examine in brief, the most common variants of Knapsack Problems and outline the design of this thesis.

1.1 Knapsack Problems and its Variants

A typical investment problem can be described as follows.
Given an amount of investment capital, a variety of projects with different capital requirements and expected profits are possible. Some of the projects are to be selected such that the budget is not exceeded and the total expected profit is maximum.

This decision problem is an important application model of the Knapsack Problem (KP).

The name Knapsack Problem comes from its relation to the hitch-hiker's decision making situation (which is the same as the investment example); a hitch-hiker packs his knapsack by selecting from among various possible objects those which will give him maximum utility or profit without exceeding the weight capacity of the knapsack.

Mathematical programming problems like *Linear Programming* and *Integer Programming* have been applied in a number of decision making situations. A KP can be classified as an integer programming problem. Because of its combinatorial structure, it is often treated as a combinatorial optimisation problem.

Mathematically, a KP can be formulated as:

Given a set of n items and a knapsack, with

p_j = profit (or value) of item j,

w_j = weight (or volume) of item j,

C = capacity of the knapsack,

select a subset of the items so as to

Maximise $z = \sum_{j=1}^{n} p_j x_j$ ... ... (1)

subject to $\sum_{j=1}^{n} w_j x_j \leq C$ ... ... (2)

$x_j = 0$ or 1, $j \in N = \{1,2, \ldots ,n\}$ ... ... (3)

where $x_j = 1$, if item j is selected;

$= 0$, otherwise.

In the literature this one-constraint, linear pure 0-1 discrete programming problem is called the 0-1 Knapsack Problem or simply the Knapsack Problem. It is also known as the Lorie-Savage Problem (1955).

There are different variants of the Knapsack Problem, some are described in the following.

1.1.1 0 - 1 Knapsack Problem

The problem defined by equations (1), (2), & (3) is called the 0 - 1 Knapsack Problem (0 - 1 KP). The term 0 - 1 appears because an item is either selected or rejected, and at most one piece of each item can be selected.

1.1.2 Unbounded Knapsack Problem

When it is possible to select any number of pieces of an item, the problem is called an Unbounded Knapsack Problem (UKP),
i.e., when equation (3) is replaced by

$$x_j \geq 0, \text{ integer} \qquad \dots \quad \dots \quad \dots \quad (4)$$

where x_j = number of units of item j selected, then UKP is the problem defined by equations (1), (2), & (4).
In this thesis, we will be dealing with the solution algorithms for UKP.

1.1.3 Bounded Knapsack Problem

When for each item there is an upper limit to the number of pieces that can be selected, the problem is called a Bounded Knapsack Problem (BKP),
i.e., when equation (4) is replaced by

$$0 \leq x_j \leq b_j, \quad j \in N = \{1,2, \dots ,n\} \qquad \dots \quad \dots \quad \dots \quad (5)$$

$$x_j \text{ integers,}$$

then BKP is the problem defined by equations (1), (2), & (5).
Strictly speaking, an UKP is a special case of BKP, since replacing b_j by ∞ or a value defined by the knapsack capacity and w_j, they are equivalent.

The 0 - 1 KP, UKP and BKP are generally referred to as Knapsack Problems (KP). KPs have been intensively studied as discrete programming problems. The reason for such interest basically derives from three facts:

(a) a KP can be viewed as one of the simplest *Integer Programming* problems;

(b) it appears as a subproblem in complex problems as in cutting stock problem. The solution of knapsack problems in solving cutting stock problems (Gilmore and Gomory, 1963) is particularly important because of the fact that in the column generation procedure that is used for cutting stock problem, repeated solution of Unbounded Knapsack Problems are used;

(c) it represents a great many practical situations such as capital budgeting, project selection, loading problems, journal selection in a library (Salkin and de Kluyver, 1975).

There are many other variants of KP and similarly structured mathematical models. They can be classified under two categories. The knapsack problems, where only one knapsack is to be filled with an optimal subset of items are called Single Knapsack Problems. The *0-1 knapsack problem,* the *bounded and unbounded knapsack problems,* the *subset-sum problem,* the *change-making problem* are all single knapsack problems. The knapsack problems where more than one knapsack is available are called Multiple Knapsack Problems. The *0-1 multiple knapsack problem,* the *generalised assignment problem* and the *bin-packing problem* can be called multiple knapsack problems.

These problems are discussed in detail in Taha (1975), Salkin and Mathur (1989) and Martello and Toth (1990); the last one includes computer codes for solving them.

1.2 Objective of Study

Knapsack Problems (KP) are widely used mathematical decision models, particularly in the areas of cutting stock, cargo loading, capital investment, etc. KP and its variants are in the class of *difficult* optimisation problems, which take up long computational time. The particular variant of KP studied in this research, namely the Unbounded Knapsack Problem (UKP), was first introduced some decades ago and the density-ordered greedy heuristic algorithm has been available since. In the 1990s, a new algorithm, called the total-value greedy heuristic appeared, but not fully explored in terms of computational time and quality. A detailed report on the exact (optimal) solution algorithms was published only lately (in 1990).

In this research all the available heuristic algorithms for UKP have been studied. The aims of this research is to generate test problems and investigate the performance of the algorithms in solving different instances of these problems by varying parameters such as problem size, knapsack capacity and profit/weight ratios. In particular, an extensive study of the performance of the *density-ordered greedy heuristic*, the *weight-ordered greedy heuristic*, the *value-ordered greedy heuristic*, the *extended greedy heuristic* and the *total-value greedy heuristic* are undertaken by comparing the quality of their solutions and the computational time with respect to the Martello-Toth exact algorithm. The results are expected to be useful in developing algorithms and software in related areas. In the column generation technique for solving linear programming problems, the ease of solving KPs will be helpful. Although, it has been the practice to take the column corresponding to the optimal solution of the KP, it is not absolutely necessary to do so. Gilmore and Gomory (1963) observed that accepting any feasible solution could be a viable option.

1.3 Organisation of the Thesis

Chapter 2 describes in brief the existing algorithms for the knapsack problem. Algorithms for the variants of KP are outlined in Section 2.1 and a detailed discussion of the solution algorithms for the unbounded knapsack problem is given in Section 2.2. Section 2.3 gives a brief description of the meta-heuristics and their likely use in finding solutions for knapsack and related problems. Section 2.4 explains Martello-Toth's exact algorithm for UKP. Dominance criterion, an important phenomenon used to greatly reduce the size of the original problem is discussed in Section 2.5. Chapter 3 is a detailed description of the computational analysis of the heuristics for UKP. Section 3.1 describes the computational design and data generation for the heuristics and gives a few sample datasets for the five problem classes generated. Section 3.2 describes the performance measures and the factors that are to be recognised in an experimental study of heuristics. Section 3.3 and 3.5 reports the computational results obtained on the five heuristic algorithms for UKP. Sections 4.1 describes the complementary effect of heuristics. Based on the behaviour of the existing heuristics, a new complementary heuristic is developed and is discussed in Section 4.2. Summary and conclusion is given in Chapter 5. The FORTRAN codes for data generation are given in Appendix A and the code for the heuristic algorithms is given in Appendix B; in Appendix C, the effect of varying the density ratio range is explained for the Class IV problem instances and in Appendix D some difficult problem instances of Class V Unbounded Knapsack Problem are described.

CHAPTER 2

2. LITERATURE REVIEW

The sections in this chapter describe the solution algorithms for Knapsack Problem in brief and discusses the heuristic algorithms for the Unbounded Knapsack Problem at length. The dominance criterion which is an important phenomenon in any solution algorithm for UKP is described in Section 2.5.

2.1 Exact Versus Heuristic Algorithms

The *time-complexity* or, simply, *complexity* of an algorithm for solving some problem is said to be the maximal number of computational steps that it takes to solve any instance of the considered problem of a given size. For example, the *time-complexity* of a given algorithm as a function of the size s is the *order* of f(s), when $s \rightarrow \infty$ and is denoted by O(f(s)) or simply $O(s)$.

An algorithm can be classified as good or bad depending on whether or not it has polynomial time complexity. Similarly, a problem can be classified as 'hard' or 'easy' depending on whether or not it can be solved exactly by an algorithm with polynomial time complexity. Based on this distinction, an elegant theory of the complexity of problems has been developed (Garey and Johnson, 1979).

Mathematical decision problems can be grouped into two classes, *viz.*, easily solvable problems (class P) for which polynomial algorithms are known and problems which require considerable computing time (class NP), for which only exponential time exact algorithms are known. A problem type is in the class P if there exists an algorithm that, for any instance, has running time (the number of computational steps required) that is bounded by a polynomial function of the problem size, i.e., problems for which polynomial-time algorithms have been devised. A problem type is in the class NP if it is possible to devise an algorithm for each problem, but no polynomial-time algorithm is known for any of them. For example, the problem of finding the maximal number among n numbers requires n-1 comparisons, thus such a problem is in P. For a 0 - 1 knapsack problem with n items to be solved exactly, we have to check, in the worst case, all 2^n combinations of items. Such a problem is in NP.

A problem is termed NP-Complete if (i) it belongs to NP and (ii) it has a property that if an efficient algorithm is found for it then an efficient algorithm can be found for every problem in NP. In this sense the NP-Complete problems are the hardest in NP. KPs are NP-Complete problems (Garey and Johnson, 1979) and are difficult to solve optimally. Obviously, the difficulty rises rapidly if the number of items go up.

An **exact algorithm** guarantees an optimal solution to a mathematical programming problem. The two principal approaches for finding an optimal solution to an integer-programming problem are the *branch-and-bound* algorithm and the *dynamic programming* algorithm. Although there are noticeable differences among different problems, in general the NP-Complete problems require a lot of computational time.

Heuristic algorithms give near - optimal solutions in reasonably short computational time. Heuristic solutions to different combinatorial problems can be found using a number of heuristics. For a KP, the use of a heuristic algorithm may be justified for several reasons. First, it is often the case that obtaining exact solutions to a knapsack problem may not be necessary and that one would be content with a solution that is *sufficiently* close to the optimal. This may well be the case, for instance, when the profit $p_{j,}$ themselves are only estimates of expected returns or when the knapsack problem is only a sub-optimisation of a much larger problem. Further as we see that there is no known polynomial algorithm for this problem and that there may well not be any such algorithm, restrictions on computing time may force one to be satisfied with a heuristic solution to large problem instances.

Although this study deals with UKP, solution methods for very closely related problems of 0 - 1 KP and BKP are briefly discussed in the following.

2.2 Solution Algorithms for Knapsack Problems

Following the notation of complexity as $O(s)$, it can be said that for a KP, *s* is the number of possible items. In the discussion below, *n* is the number of items and *C* is the knapsack capacity.

2.2.1 0 - 1 Knapsack Problem

This problem can be solved **exactly** by *reduction algorithms* where the number of variables are first reduced before applying the algorithm (Ingargiola and Korsh, 1973; Martello and Toth, 1988, 1990 and Nauss, 1996). It can also be solved **heuristically** by a method of *relaxation and upper bounds* (Dantzig, 1957), where the computation for the Dantzig bound requires $O(n)$ time if the items are sorted according to non-increasing values of the profit per unit weight. Other *heuristic algorithms* by 0-1 KP include Sahni (1975) and Balas and Zemel (1980), which require $O(n \log n)$.

2.2.2 Bounded Knapsack Problem

This problem can be solved **exactly** by *branch-and-bound* algorithms (Martello and Toth, 1977; Ingorgiola and Korsh, 1977 and Bulfin *et al.*, 1979). Aittoniemi and Oehlandt (1985) gives an experimental comparison of these, indicating the Martello and Toth (1977) one as the most effective. It can also be solved by *dynamic programming* (Gilmore and Gomory, 1966; Nemhauser and Ullmann, 1969) method requiring $O(nC^2)$ time in the worst case and the space complexity is $O(nC)$ and can only solve problems of very limited size. The **heuristic** solution algorithms are *upper bounds and approximate algorithms,* where the computation time is $O(n)$.

2.2.3 Unbounded Knapsack Problem

The solutions to this problem include **exact** algorithms based on *branch - and - bound* (Martello and Toth, 1977; Cabot, 1970; Gilmore and Gomory, 1963) and *dynamic programming* (Garfinkel and Nemhauser, 1972). The **heuristic** algorithms are *upper bounds and approximate algorithms* (Magazine *et al.*, 1975; Hu and Lenard, 1975), the time complexity for the computation of the upper bounds is $O(n)$ and the time complexity of the approximate (Greedy) algorithm is $O(n)$, plus $O(n \log n)$ for the preliminary sorting.

This research focuses on the solution of the Unbounded Knapsack Problem (UKP). In the literature, there are five main heuristic algorithms for UKP:

a) *Density - ordered greedy heuristic* (Dantzig, 1957; Martello and Toth, 1990),

b) *Weight - ordered greedy heuristic* (Horowitz and Sahni, 1978; Kohli and Krishnamurti, 1995),

c) *Value - ordered greedy heuristic* (Horowitz and Sahni, 1978; Kohli and Krishnamurti, 1995),

d) *Extended greedy heuristic* (White, 1991),

e) *Total - value greedy heuristic* (White, 1992; Kohli and Krishnamurti, 1992; Lai, 1993).

2.3 Heuristic Algorithms for the Unbounded Knapsack Problems

The aforementioned five heuristic algorithms are discussed in the following.

The solution method of these five greedy heuristics is termed 'greedy' because at each step (except possibly the last one) we choose to introduce that object which according to one criterion or the other would increase the objective function value the most. An object once selected, stays in the knapsack and therefore in the solution. The items can be ordered (a) in decreasing order of density p_i/w_i, (b) in increasing order of the item weights w_i, (c) in decreasing order of the profit of the item p_i and (d) in decreasing order of the total-value $\lfloor C/w_i \rfloor p_i$.

2.3.1 Density - ordered Greedy Heuristic (H_1)

This is the classic heuristic for the unbounded knapsack problem. This procedure has been discussed in the literature among others, by Garey and Johnson (1979) and Martello and Toth (1990). Dantzig (1957) first introduced this algorithm.

Density-ordered greedy heuristic recursively determines a solution by making a variable with smallest marginal unit cost as large as possible.

First, order the items so that

$\rho_1 \geq \rho_2 \geq \dots \quad \dots \quad \dots \geq \rho_{n-1} \geq \rho_n$

where, $\rho_j = p_j / w_j$, $\quad 1 \leq j \leq n$

Then set

(a) $\quad x_1 = \lfloor C / w_1 \rfloor;$

(b) $\quad x_j = \lfloor (C - \sum_{k=1}^{j-1} x_k w_k) / w_j \rfloor, \qquad 2 \leq j \leq n$

where for $z \in Z_+$, $\lfloor z \rfloor$ is the integer part of z.

H_1 gives good results, but the worst case result is poor and it can be shown that there are instances where the optimal solution is almost twice the greedy solution. Under some restrictive assumptions, the greedy algorithm will give optimal solution (Magazine *et al.*, 1975; Hu and Lenard, 1975; White, 1991). An example of H_1 is as follows.

Example 1

$C = 100$

$w_2 = 50, \quad w_1 = 51$

$p_2 = 99, \quad p_1 = 102$

$\rho_1 = 102/51 > \rho_2 = 99/50$

$H_1 \rightarrow x_2 = 0, \; x_1 = 1, \; z(H_1) = 102$

Optimal $\rightarrow x_2 = 2, \; x_1 = 0, \; z(opt) = 198$

The heuristic solution value is almost half of the optimal solution value.

Specifically, $z(H_1) = 0.52 \, z(opt)$

2.3.2 Weight - ordered Greedy Heuristic (A)

Horowitz and Sahni (1978) formulated a greedy approach attempting to obtain a solution. This method tries to be greedy with the capacity and thus requires the objects to be ordered in non-decreasing weights, we try to put as many objects as possible with the least weight into the knapsack, thus using up as much capacity. This heuristic has arbitrarily bad worst-case bounds (Horowitz and Sahni, 1978; Kohli and Krishnamurti, 1995) though the capacity is used up slowly with the profits coming in rapidly enough. Example 2 shows a very bad instance for Heuristic A.

Example 2

$C = 100$

$w_2 = 10, \quad w_1 = 9$

$p_2 = 1000, \quad p_1 = 1$

$A \rightarrow x_2 = 0, \; x_1 = 11$

$z(A) = 11$

Optimal $\rightarrow x_2 = 10, \; x_1 = 0$

$z(opt) = 10000$

$z(A) = 0.0011 \; z(opt)$

It is possible to find randomly generated instances where the weight-ordered greedy heuristic provides the optimal solution value (Example 3). In general they are expected to perform poorly.

Example 3

$C = 20$

$w_3 = 10, \quad w_2 = 15, \quad w_1 = 18$

$p_3 = 15, \quad p_2 = 24, \quad p_1 = 25$

$A \rightarrow x_3 = 2, \; x_2 = 0, \; x_1 = 0$

$z(A) = 30$

Optimal → $x_3 = 2,\ x_2 = 0,\ x_1 = 0,\quad z(opt) = 30$

2.3.3 Value - ordered Greedy Heuristic (B)

This greedy heuristic discussed by Horowitz and Sahni (1978) followed by Kohli and Krishnamurti (1995) considers objects in order of non-increasing profit values. This method too has arbitrarily bad worst-case bounds (Horowitz and Sahni, 1978; Kohli and Krishnamurti, 1995) and does not usually yield an optimal solution though the objective function value takes large increases at each step. The number of steps is reduced as the knapsack capacity is used up at a rapid rate. A bad instance is given in Example 4.

Example 4

$C = 100$

$w_2 = 99,\quad w_1 = 1$

$p_2 = 2,\quad p_1 = 1$

B → $x_2 = 1,\ x_1 = 1$

$z(B) = 3$

Optimal → $x_2 = 0,\ x_1 = 100$

$z(opt) = 100$

$z(B) = 0.03\ z(opt)$

There are of course instances where the value-ordered greedy heuristic gives the optimal solution value (e.g., Example 5). In general, this heuristic too is expected to perform poorly.

Example 5

$C = 80$

$w_4 = 20,\quad w_3 = 18,\quad w_2 = 15,\quad w_1 = 7$

$p_4 = 36, \quad p_3 = 20, \quad p_2 = 20, \quad p_1 = 9$

$B \rightarrow x_4 = 4, \ x_3 = 0, \ x_2 = 0, \ x_1 = 0, z(B) = 144$

$\text{Optimal} \rightarrow x_4 = 4, \ x_3 = 0, \ x_2 = 0, \ x_1 = 0, \ z(opt) = 144$

2.3.4 Extended Greedy Heuristic (H_2)

White (1991) discussed an extension of H_1, which he called H_2. This involves pairs of items rather than a single item as in the density-ordered greedy heuristic. It requires that the best combination of the first two items (from the ratio sorted list in non - increasing order) be taken and then the best combination of the next two items is taken, and so on. Unfortunately, neither H_2 is always superior to H_1 nor indeed is H_1 always superior to H_2. Although the worst case result with a ratio bound of 2 is the same for both heuristics, on many occasions the two-at-a-time heuristic (H_2) can be better. It is possible that H_1 gives an optimal solution, with H_2 not giving an optimal solution and also it is possible that H_2 gives an optimal solution, but H_1 does not give an optimal solution. If H_1 uses up exactly the available resources, then H_1 definitely gives the optimal solution. But this need not be true with H_2. Example 6 is an instance where H_2 is better than H_1.

Example 6

$C = 10$

$w_3 = 3, \ w_2 = 2, \ w_1 = 1$

$p_3 = 14, \ p_2 = 8, \ p_1 = 1$

$\rho_3 = 14/3 > \rho_2 = 4 > \rho_1 = 1$

$H_1 \rightarrow x_3 = 3, \ x_2 = 0, \ x_1 = 1, \ z(H_1) = 43$

$H_2 \rightarrow x_3 = 2, \ x_2 = 2, \ x_1 = 0, \ z(H_2) = 44$

Example 7 is an instance where H_2 is worse than H_1.

Example 7

$C = 10$

$w_4 = 3, \; w_3 = 2, \; w_2 = 1, \; w_1 = 1$

$p_4 = 20, \; p_3 = 12, \; p_2 = 5, \; p_1 = 1$

$\rho_4 = 20/3 > \rho_3 = 6 > \rho_2 = 5 > \rho_1 = 1$

$H_1 \rightarrow \; x_4 = 3, \; x_3 = 0, \; x_2 = 1, x_1 = 0, z(H_1) = 65$

$H_2 \rightarrow \; x_4 = 2, \; x_3 = 2, \; x_2 = 0, x_1 = 0, z(H_2) = 64$

If combinations of 3 or more items are considered instead of 2, we might call them H_3, H_4, and so on. These, however, clearly increases computation time requirements and lose the benefits of obtaining solutions quickly. It may be noted that H_n, where n = number of items, is in fact an exact algorithm for the original problem.

2.3.5 Total-Value Greedy Heuristic (TV)

Total-Value Heuristic (White, 1992; Kohli and Krishnamurti, 1992; Lai, 1993) is another heuristic solution method for the unbounded knapsack problem.

At step i, the total-value heuristic selects an item for which the values $p_j \lfloor C_i/w_j \rfloor$ across all available items j is maximum, where C_i is the available knapsack capacity at the beginning of step i. The items need not be sorted in a non-increasing order, because all the items have to be considered at every step.

Example 8 is an instance where the total-value greedy heuristic gives the optimal solution value. The density-ordered greedy heuristic and the extended greedy heuristic in this instance performs poorly.

Example 8

$C = 30$

$w_4 = 12,\ w_3 = 10,\ w_2 = 9,\ w_1 = 8$

$p_4 = 22,\ p_3 = 21,\ p_2 = 20,\ p_1 = 19$

$\rho_4 = 22/12 < \rho_3 = 21/10 < \rho_2 = 20/9 < \rho_1 = 19/8$

$H_1 \rightarrow x_4 = 0,\ x_3 = 0,\ x_2 = 0, x_1 = 1, z(H_1) = 57$

$H_2 \rightarrow x_4 = 0,\ x_3 = 0,\ x_2 = 2, x_1 = 1, z(H_2) = 59$

$TV \rightarrow x_4 = 0,\ x_3 = 3,\ x_2 = 0,\ x_1 = 0,\ z(TV) = 63$

Optimal $\rightarrow x_4 = 0,\ x_3 = 3,\ x_2 = 0,\ x_1 = 0,\ z(opt) =63$

Lai (1993) called this solution method as *Heuristic A*. He showed that this heuristic has a worst-case performance ratio $\geq 4/7$.

White (1992) and independently Kohli and Krishnamurti (1992) showed that the worst case bound of TV given by $1/\sum_{i=1}^{\infty} 1/h(i)$ where $h(i\ )$ is an integer value given by the recursion $h(1) = 1$, $h(2) = k + 1$, $h(i\ ) = [h(i - 1)].[h(i - 1)+1]$ for $i \geq 3$, is always better than that of H_1 given by $k/(k + 1)$ (Fisher, 1980) where k is the integer part of the ratio of the knapsack capacity to the weight of the heaviest item, i.e., $k = \lfloor C/w_{max} \rfloor$. H_1 behaves like TV with the integrality constraint removed (i.e., if the fractional amounts of items are allowed to be included in the knapsack). The performance of the heuristics depend on k. As k increases, the difference between $\lfloor 1/w_i \rfloor$ and $1/w_i$ for $i = 1, 2, \ldots, n$ is reduced, and consequently the ordering due to the total-value heuristic tends to be the ordering due to the density-ordered greedy heuristic. As a result the difference between the two worst-case bounds decreases.

The principal benefit of the total - value heuristic appears to be in the fact that it considers all three parameters -- unit weight, unit value and knapsack capacity -- in ordering items, i.e., it selects items in a non-increasing order of their maximum possible contribution to the solution value given the available knapsack capacity at each step. A consequence of considering all three parameters is that TV *always gives a better worst-case performance than that of* H_1 for the unbounded knapsack problem as shown in Table 2.1. Individual problem instances do however exist where H_1 gives better results than TV (e.g., Example 9).

Table 2.1: Worst-case Bounds for Density - ordered Greedy Heuristic and Total - value Heuristic. (From Kohli & Krishnamurti, 1992)

† k	r(TV)	r(H_1)
1	0.5913555	0.5000000
2	0.7026825	0.6666667
3	0.7678212	0.7500000
4	0.8101038	0.8000000
5	0.8396093	0.8333333

We observe from the above table that the difference between the worst-case bound for TV and H_1 is the largest for k=1. This difference decreases as k increases. As k approaches infinity, both heuristics obtain the optimal solution. Because of the greater number of operations needed per step, TV takes more computational time than H_1.

† k = number of the largest item that can fit into the knapsack and
r = (Total profit by heuristic algorithm) / (Total profit by optimal algorithm).

Example 9

$C = 41$

$w_7 = 7,\ w_6 = 8,\ w_5 = 5, w_4 = 4,\ w_3 = 9,\ w_2 = 9,\ w_1 = 3$

$p_7 = 42,\ p_6 = 46,\ p_5 = 26, p_4 = 20,\ p_3 = 38,\ p_2 = 32,\ p_1 = 10$

$\rho_7 = 42/7 > \rho_6 = 46/8 > \rho_5 = 26/5 > \rho_4 = 20/4 > \rho_3 = 38/9 > \rho_2 = 32/9 > \rho_1 = 10/3$

$H_1 \rightarrow x_7 = 5,\ x_6 = 0,\ x_5 = 1,\ x_4 = 0,\ x_3 = 0,\ x_2 = 0,\ x_1 = 0,\ z(H_1) = 236$

$TV \rightarrow x_7 = 0,\ x_6 = 5,\ x_5 = 0,\ x_4 = 0,\ x_3 = 0,\ x_2 = 0,\ x_1 = 0,\ z(TV) = 230$

2.3.6 Combination of Greedy Heuristics

Kohli and Krishnamurti (1995) analysed the worst-case performance of a combination of greedy heuristics (density-ordered greedy, weight-ordered greedy, value-ordered greedy and the total-value greedy heuristic) for the Unbounded Knapsack Problem.

An analysis of composite heuristics provides insight into why one heuristic performs well while the other performs poorly. If the heuristics complement each other, the composite solution value can be closer to the optimal than the solution value of the individual heuristics. This was shown by the composite of the density-ordered and total-value greedy heuristics by guaranteeing a tight worst-case bound of (k+1)/(k+2). The density-ordered greedy heuristic by itself performs most poorly when the densest item leaves a significant capacity of the knapsack unused, also leaving an insufficient amount of the weight capacity to fit any other item. The total-value greedy heuristic compensates for this limitation by choosing items with lower density that fill more of the knapsack and hence contribute more to the total solution value. But this heuristic cannot discriminate between the items that have the same total-value contribution with different densities. Here the density-ordered greedy heuristic is better than the total-value heuristic by choosing items that fill the knapsack at a more rapid rate. The density-ordered and total-value greedy heuristics appear to complement each other in this sense. However, the weight-ordered and value-ordered greedy heuristics use very little information regarding the problem so much so that they seem to neither complement each other, nor the density-ordered and total-value heuristics. The usefulness of the weight-ordered and the value-ordered greedy heuristics thus seem insignificant in solving UKPs. A combination of the density-ordered and the total-value greedy heuristics can be used to provide better lower bounds on the optimal solution value.

2.4 Meta - Heuristics

Meta-heuristics (Osman and Kelly, 1996; Reeves, 1993) are recent development in approximate search methods for solving complex optimisation problems that arise in business, commerce, engineering, industry and many other areas. This class of approximate methods developed in the early 1980s, was designed to attack hard combinatorial optimisation problems where classical heuristics have failed to be effective and efficient. They provide general frameworks that allow for creating new hybrids by combining different concepts derived from classical heuristics, artificial intelligence, biological evolution, neural systems and statistical mechanics. The approaches include genetic algorithms, greedy search procedure, problem-space search, neural networks, simulated annealing, tabu search, threshold algorithms and their hybrids.

A meta-heuristic can be defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search spaces using learning strategies to structure information in order to find near-optimal solutions efficiently.

Meta-heuristics have not been used in the solution algorithms for the Unbounded Knapsack Problem but the different search methods can definitely be incorporated in the heuristic approach because of the generation process being iterative.

Classification of a comprehensive list of references on the theory and application of meta-heuristics is provided by Osman and Laporte (1996).

For completeness, a brief description of the most popular meta-heuristics is given in the following.

2.4.1 Simulated Annealing

Simulated Annealing came to use in the early 1980s as a heuristic technique for combinatorial optimisation problems and was said to be the most simple and robust algorithm capable of providing good quality solutions to some very difficult problems.

The algorithm was first published by Metropolis *et al.* (1953) and later by Kirkpatrick *et al.* (1983).

This algorithm is based on the analogy between the annealing process of solids and the problem of solving combinatorial optimisation problems. In condensed matter physics, annealing denotes a process in which a solid in a heat bath is melted by increasing the temperature of the heat bath to a high maximum value at which all molecules of the solid randomly arrange themselves into a liquid phase.

This approach is regarded as a variant of the well-known heuristic technique of local (neighbourhood) search, in which a subset of the feasible solutions is explored by repeatedly moving from the current solution to a neighbouring solution. However, this technique needed disappointingly long running times even to find the approximate convergence to optimum. But a number of experiments and practical applications shows that annealing can provide a useful solution method for a variety of problems, generally out-performing standard descent methods and sometimes competing effectively with specialist heuristics. This method is easy to implement, it is applicable to almost any combinatorial optimisation problem and it usually provides reasonable solutions. When faced with the challenge of designing a heuristic solution for a new problem, simulated annealing is certainly worth considering.

Simulated annealing is applicable in Knapsack Problems (Cagan, 1994; Drexl, 1988; Hanafi *et al.*, 1996; Abramson *et al.*, 1996 and Ohlsson *et al.*, 1993) as well as many other applications.

2.4.2 Tabu Search

Tabu search is an iterative meta-heuristic search procedure introduced by Glover (1986) for solving optimisation problem. This search is based on intelligent problem solving. It shares the ability to guide a subordinate heuristic (such as the local neighbourhood search procedure) to continue the search beyond a local optimum where the embedded heuristic will normally become trapped. The process in which the tabu search method seeks to transcend local optimality is based on an aggressive evaluation that chooses the best available move at each iteration even when this move may result in a degradation of the objective value. This search begins in the same way as an ordinary local search, proceeding iteratively from one solution to another until a chosen termination criterion is satisfied. Many tabu search implementations are largely or wholly deterministic. An exception occurs for the variant called probabilistic tabu search, which selects moves according to probabilities based on the status and evaluations assigned to these moves by the basic tabu search principles.

Tabu search concepts and strategies offer a variety of fruitful possibilities for creating hybrid methods in combination with other approaches.

A tabu search method that incorporates tabu restrictions on the logical structure of the generated problem was studied by Dammeyer and Voss (1993) and Hanafi *et al.* (1996) on Knapsack Problems. It finds use in Cutting and Packing Problems (Laguna and Glover, 1993). Tabu search is also applicable in production scheduling, routing, design, network planning, expert systems and a variety of other areas.

2.4.3 Genetic Algorithms

Genetic Algorithms are a class of adaptive search methods based on a abstract model of natural evolution. It can also be understood as the intelligent exploitation of a random search. They were first developed in the early 1970s by Holland (1975), and later refined by De Jong (1975), Goldberg (1989), and many others. Only recently their potential for solving combinatorial optimisation problems has been explored. The most early applications were in the realm of Artificial Intelligence -- game-playing and pattern recognition for instance.

The name Genetic Algorithm originates form the analogy between the representation of a complex structure by means of a vector of components, and the idea, familiar to biologists, of the genetic structure of a chromosome. For example, in the selective breeding of plants or animals, offspring are sought which have certain desirable characteristics that are determined at the genetic level by the way the parents' chromosomes combine. The basic idea is to maintain a population of candidate solutions that evolves under a selective pressure that favours better solutions. Generally, Genetic Algorithm is an iterative procedure that operates on a finite population of N chromosomes (solutions). The chromosomes are fixed strings with binary values (0 or 1) at each position. Each chromosome of the population are evaluated according to a fitness function. Members of the population are selectively interbred, often in pairs to produce offspring. The fitter a member of the population the most likely it is to produce an offspring. Genetic operators are used to facilitate the breeding process that results in offspring inheriting properties from their parents. The offspring are evaluated and placed in the population replacing the weaker members of the last generation. The new chromosomes resulting from these operations form the population for the next generation and the process is repeated until the system ceases to improve.

Genetic Algorithms encounters a number of problems when solving combinatorial problems. They fail to find satisfactory solutions for many reasons. The genetic algorithm binary encoding/decoding has been found unsuitable and normal cross-over operations often lead to many infeasible solutions. This can be overcome by using genetic algorithm in combination with other techniques such as the branch and bound, local search, simulated annealing and tabu search.

Genetic Algorithm is applicable in many combinatorial problems like the Bin Packing and related problems (Falkenauer and Delchambre, 1992; Reeves, 1996) and the Knapsack Problems (Fairley and Yates, 1993; Thiel and Vob, 1994).

2.4.4 Neural Networks

Neural Networks are models based on the functioning of the human brain. They have been successful in solving problems whose structures can be exploited by a process linked to those of associated memory. They have been successfully used to solve a variety of practical problems in areas such as pattern recognition and optimisation.

The interest in using neural networks in combinatorial optimisation problems was pioneered by the work of Hopfield and Tank (1985) and later developed by Aarts and Korst (1989). This is best used in any class of problems because of its robustness, generalisation capabilities, and speed of operation through hardware implementability of inherent parallel structures.

The networks consist of a set of competing connected elements. The competing elements are logic units with binary states and are linked by symmetric connections. Each connection is associated with a weight representing the interconnections between units when both are 'on'. A consensus function assigns to each configuration of the network a real value. The units may change their state in order to maximise the consensus. A state change of an individual unit is determined by a deterministic response function of the states of its adjacent units. If the response function is a probability function, then the randomised version of the network is called the Boltzmann machine. The challenge of the model is to choose appropriate network structure and corresponding connection strengths such that the problem of finding near optimal solutions of the optimisation problem is equivalent to finding maximal configurations of a network.

Neural networks find their application in Knapsack Problems (Glover, 1994; Ohlsson *et al.*, 1993) and many other combinatorial problems, like the

Cutting and Packing Problems (Bahrami and Dagli, 1994) and the Assignment Problems (Kurokawa and Kozuka, 1994).

2.5 Martello - Toth Exact Algorithm

An UKP being an NP-Complete problem requires a lot of computational time. The various approaches to its exact solution include branch-and-bound algorithm proposed by Gilmore and Gomory (1963), Cabot (1970) and Martello and Toth (1978).

Many instances of UKP can be solved by branch-and-bound algorithms for very large values of n . For these problems, the preliminary sorting of the items requires, on average, a comparatively high computing time. This was overcome by Balas-Zemel algorithm (1980) which is based on the "core problem". The idea of Balas-Zemel algorithm is to first solve, without sorting, the continuous relaxation of UKP, thus determining the Dantzig upper bound, and then searching for heuristic solutions of approximate core problems giving the upper bound value for UKP. When such attempts fail, the reduced problem is solved through two effective exact procedures, the Fayard-Plateau algorithm (1982) and the Martello-Toth algorithm (1988).

Martello-Toth algorithm is easily the best of the two. The procedure can be sketched as follows:

Step 1: Choose a cut off value for p_j / w_j to select a core (which is a subset of the original problem). This would be a very small fraction of n.

Step 2: Solve the core problem optimally. This solution is an approximate solution to the original problem.

Step 3: If this solution value equals that of the upper bound computed for the original problem, the optimal value is found.

Step 4: Otherwise, include a variable not in the core that has the potential to improve the existing solution.

Step 5: The core with the new variable is solved again and the process continued until an optimum is found.

Martello-Toth algorithm is an improvement over the Fayard-Plateau algorithm in many respects. Here, the approximate solution determined, is more precise (often optimal). This is obtained through a more careful definition of the approximate core and through exact (instead of heuristic) solution of the corresponding problem. The probability of obtaining such an approximate solution that is optimal is high because of a tighter upper bound computation. Finally, the exact solution of the subproblems are obtained by adapting an effective Martello-Toth branch-and-bound algorithm (1978).

Although the Martello-Toth (1990) algorithm is efficient, it still takes quite some time for finding the exact solution. In our study, great difficulty arose for problem instances with problem size n = 500. Problem with larger sizes were easily solved within reasonable computational time. The difficult instances are further investigated (see Appendix D) but the reason for such huge running time is not clear. A sample data set is included in that appendix for further research.

2.6 Dominance Criteria

One very important aspect in any solution algorithm for UKP is the phenomenon of dominance discussed by Martello and Toth (1990); Dudzinski (1991), Johnston and Khan (1995) and Zhu and Broughan (1996).

The domination of items can be defined as follows:

Item i dominates item j if there exists positive integer r such that $rw_i \leq w_j$ and $rp_i \geq p_j$. An example follows. The implication of this is that an optimal solution to an instance of UKP obtained using only the undominated items cannot be worse than any other solution that contains one or more dominated items. The dominated items therefore can be eliminated and the problem size greatly reduced.

Martello and Toth (1990) and Dudzinski (1991) reported that with p and w randomly generated from a uniform distribution, the number of undominated items is extremely small. For instance, Dudzinski's (1991) computational result shows that the average number of undominated items for an uncorrelated problem (items are defined to be uncorrelated if there is no relation between the profits and the corresponding weights of the items) of size 500 is in fact 2.3 and our computation yields on average 2.2 undominated items. A theoretical analysis supporting this result in regard to item dominance is available in Johnston and Khan (1995). Hence, a knapsack problem can be reduced to a very small size and solving it would take negligible computational time. If p and w are correlated, there are many undominated items and by maintaining a very strong correlation, problem instances with a large number of undominated items can be constructed. In this research, the dominance phenomenon is studied on five different classes with varying correlation.

Example 10

$w_4 = 15, \; w_3 = 13, \; w_2 = 11, \; w_1 = 5$

$p_4 = 60, \; p_3 = 55, \; p_2 = 39, \; p_1 = 20$

Item 1 dominates item 2 (where r = 2) and hence item 2 can be eliminated. Similarly, item 1 dominates item 4 (r = 3) and hence item 4 can be eliminated. The remaining undominated items are items 1 and 3. The capacity of the knapsack should obviously be such that it must accommodate at least one unit of the largest item.

CHAPTER 3

3. EMPIRICAL ANALYSIS OF HEURISTICS FOR THE UNBOUNDED KNAPSACK PROBLEM

The heuristic methods have always been helpful in solving problems that were too large or complex for developing algorithms. The effectiveness of a heuristic for solving a given class of problems can be demonstrated by empirical testing.

3.1 Computational Design and Data Generation

We analyse the experimental behaviour of exact and approximate algorithms for the Unbounded Knapsack Problem on a set of randomly generated test problems. The heuristics are evaluated by experimenting with a series of problem instances using different selection of problem classes and setting various performance parameters. They are often used to identify "good" approximate solutions to difficult problems in less time than is required for an exact algorithm to uncover an exact solution. The computational testing is done to compare the performance of the five heuristics - the density-ordered greedy, the weight-ordered greedy, the value-ordered greedy, the extended greedy and the total-value greedy with the optimal solution algorithm (obtained by Martello-Toth algorithm). We compare the FORTRAN 77 implementations of the five heuristics. All runs have been executed on a 200 MHz Pentium Pro with option "-o" for the FORTRAN compiler.

A set of 2500 test problems was randomly generated with size n equal to 50, 100, 500, 1000, 5000, 10000, 20000, 30000, 40000 and 50000 (250 problems of each size). These test problems were generated with varying degrees of correlation between the constraint and the objective function coefficients, i.e., between the profits and weights of the items, as described in sections 2.10 and 3.5 of Martello and Toth (1990) with an additional two types of problem instances with stronger correlation. All the data sets were randomly generated as described in the following. The FORTRAN codes for data generation are given in Appendix A. The sample data sets are shown in Table 3.1.

Uncorrelated : w_j uniformly random in [10, 9999]
(Class I) p_j uniformly random in [1, 9999]

Weakly Correlated : w_j uniformly random in [10, 9999]
(Class II) p_j uniformly random in
$[w_j - 100, \ w_j + 100]$

Strongly Correlated : w_j uniformly random in [10, 9999]
(Class III) $p_j = w_j + 100$

Very Strongly Correlated : w_j uniformly random in [1, 9999]
(Class IV) p_j uniformly random in [1, 9999]
$2.0 \le p_j/w_j \le 2.5$ †

Very Very Strongly Correlated : w_j uniformly random in [1, 99999]
(Class V) $p_j = w_j * \{(w_j - \min w_j + 1)/(\max w_j - \min w_j + 1)\} * 100$

The right hand side C of the knapsack constraints, the knapsack capacity is the integer value $\in$ [100000, 1000000] satisfying the condition $\max w_j \le C$.

† The p_j/w_j ratio is limited to the range (2.0, 2.5) so as to get a large number of undominated items. Investigation on different ratio ranges is given in Appendix C.

Table 3.1: Sample data sets for the five problem classes

Class I			Class II			Class III			Class IV			Class V		
Ratio	w_j	p_j	Ratio	w_j	p_j	Ratio	w_j	p_j	Ratio	w_j	p_j	Ratio	w_j	p_j
1.502	3262	4898	1.012	8375	8474	1.024	4254	4354	2.129	3836	8168	35.830	35830	1283801
0.227	1791	406	0.746	118	88	1.028	3561	3661	2.287	1211	2769	92.717	92716	8596343
6.914	570	3941	0.987	5729	5652	1.095	1052	1152	2.427	724	1757	14.031	14031	196870
0.047	2912	136	0.994	5698	5662	1.022	4570	4670	2.095	569	1192	55.743	55742	3107201
0.473	4953	2341	1.009	3365	3395	1.075	1333	1433	2.351	2016	4740	18.894	18894	356986
0.636	3506	2229	0.998	3802	3795	1.030	3330	3430	2.019	4718	9524	47.336	47336	2240719
0.688	2982	2053	1.001	3807	3811	1.077	1294	1394	2.443	3367	8224	91.378	91377	8349839
0.299	4335	1295	0.995	8134	8097	1.031	3212	3312	2.245	4270	9585	47.973	47973	2301431
1.917	2378	4558	1.012	8202	8301	1.032	3088	3188	2.309	4103	9474	65.183	65182	4248735
2.525	1247	3149	0.986	1231	1214	1.030	3307	3407	2.016	3717	7495	91.399	91398	8353678

By increasing the correlation, we can decrease the difference between $\max_j \{ p_j/w_j \}$ - $\min_j \{ p_j/w_j \}$. This will increase the expected difficulty of the corresponding problems. By identifying the dominated items and then applying the heuristic algorithms to the undominated items, the computational time can be greatly decreased. The dominance phenomenon on the above five classes have been extensively studied in Section 3.2. The first three problem classes (*Class I, II and III*) discussed by Martello and Toth (1990) gives very few undominated items whereas the fourth and the fifth classes are generated with stronger correlation so as to give a large number of undominated items. The *Class IV* is in fact similar to the value-independent (sum-of-subset in Martello-Toth's terminology) knapsack problem, where the density ratios are same for all items; in this class of problem, the ratio range is narrow, but not constant. The problem instances in *Class V* have been generated where the ratios are proportional to the respective item weights. The reason for this scheme is as follows.

As evident from the definition of the dominance criterion, items of higher weight are usually dominated by those of lower weight (the items with the lowest weight is never dominated). This is avoided for this class of problems by ensuring that items of higher weight have proportionally higher density ratios. Indeed, it is found that, for this class of problems, only a small fraction of items are dominated.

Problem instances with smaller size n are generated with smaller w and p range and large size problem instances are generated with larger w and p range to obtain data sets with n as large as possible. Taking large w range for smaller problem size would give fewer undominated items. Problem instances in *Class V* are generated with w in the range [1, 99999] so as get large number of undominated items. Despite these minor modifications in the data generation for different n, the results should be comparable because the problem instances were unchanged across different algorithms.

3.2 Effect of Dominance on the Five Problem Classes

To study the dominance phenomenon on the five problem classes, the number of undominated items is found for 2500 test problems.

Table 3.2 shows the average number of undominated items for the five problem classes. The first three types of correlation discussed by Martello and Toth (1990) results in few undominated items (Figure 3.1a) where as the fourth and the fifth classes are generated with stronger correlation so as to give many undominated items (Figure 3.1b).

Table 3.2: **Average number of Undominated items (average of 5 problem instances for each class and each *n*)**

Number of items, n	Number of undominated items, N				
	Uncorrelated (Class I)	Weakly Correlated (Class II)	Strongly Correlated (Class III)	Very Strongly Correlated (Class IV)	Very Very Strongly Correlated (Class V)
50	2.0	1.8	2.4	18.0	37.8
100	1.7	2.1	2.9	26.0	97.0
500	2.2	4.6	5.8	49.8	389.8
1000	2.6	2.8	7.2	83.4	954.8
5000	3.4	3.4	8.6	173.4	4882.2
10000	2.6	2.6	6.0	241.2	9491.4
20000	2.8	4.6	5.0	285.2	18123.0
30000	2.8	4.2	6.0	377.6	25869.2
40000	3.2	3.6	6.2	399.6	32908.0
50000	1.8	3.4	6.6	401.8	39322.2

We see that as the number of items, *n*, increases, there is no well defined increase in the number of undominated items, *N*, for the Uncorrelated, Weakly Correlated and Strongly Correlated classes of problems where as, the number of undominated items increases with *n* for the Very Strongly Correlated and the Very Very Strongly Correlated classes of problems. This behaviour is illustrated in the figures 3.1a and 3.1b.

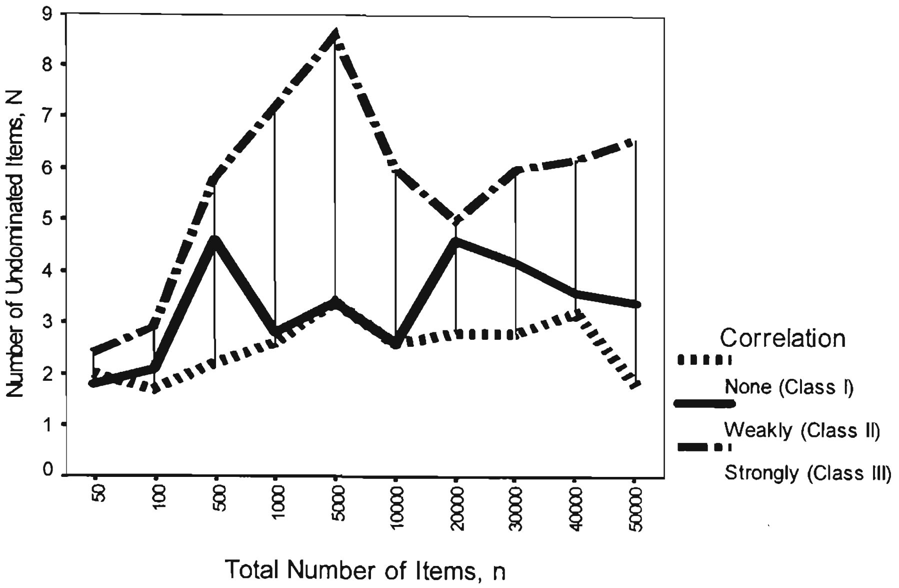

Figure 3.1a: Number of Undominated items for Uncorrelated, Weakly Correlated and Strongly Correlated class of UKP ‡

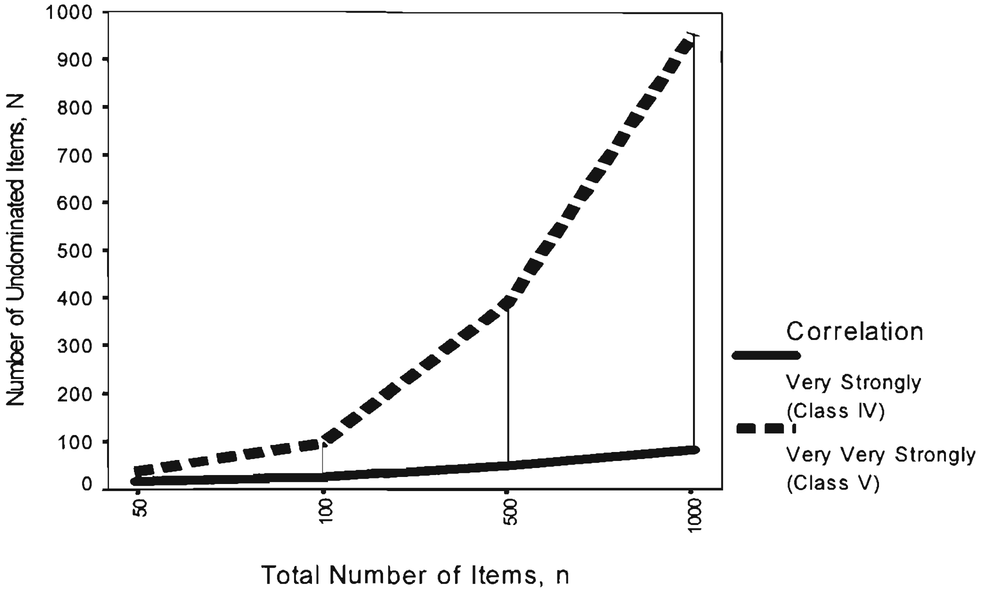

Figure 3.1b: Number of Undominated items for Very Strongly Correlated and Very Very Strongly Correlated class of UKP

‡ Note: All the five classes of problems are not shown in the same figure because of huge difference in scales.

3.3 Performance Measures and Factors

This section discusses the ways in which to obtain high-quality solution based on the attributes that are well recognised as valid criteria for the comparison of heuristic algorithms.

3.3.1 Measuring Performance

The most important decision one makes in an experimental study of heuristics is the characterisation of algorithm performance (Barr *et al.*, 1995). For the Unbounded Knapsack Problem, the following questions often arise when testing a given heuristic on a specific problem instance.

1. What is the quality of the best solution found?
2. How long does it take to determine the best solution?
3. How quickly does the algorithm find a good solution?
4. How robust is the method?
5. What is the tradeoff between feasibility and solution quality?

The quality of the solutions obtained by the heuristics of UKP has been the most important consideration in our study. The quality of a solution is judged by comparing it with the corresponding optimal solution obtained by the exact algorithm of Martello-Toth (1990).

The running time required by the heuristic algorithm to solve a given problem is often a crucial consideration in choosing between competing algorithms. This has not been particularly experimented for the heuristics of UKP in our study as all the existing heuristics take a small fraction of computation time when compared with the execution time of Martello-Toth exact algorithm. The time taken by each run has been recorded.

The algorithms tested in this study are all greedy type, and in general, they all yield good solutions quickly. Therefore, the time to find good solutions is not very important. The feasibility of the solutions is not relevant for UKP. The robustness of the algorithms have been indirectly tested by testing them on problem instances of varying structure and size.

There are other factors that need to be considered before selecting a heuristic algorithm. Ease of implementation is an important consideration. Difficult-to-code algorithms that require substantial amounts of computer time may not be worth the effort if they only marginally outperform an easy-to-code algorithm that is extremely efficient. The algorithm should also be flexible, in the sense that it should be able to handle all problem variations. A heuristic for UKP that can solve only small problems, is clearly, not as flexible as the one that can solve both small and large UKPs. Simplicity is another important consideration. Simply stated algorithms are more appealing to the user than cumbersome algorithms and they more readily lend themselves to various kinds of analysis.

The five heuristic solutions - the density-ordered greedy, the weight-ordered greedy, the value-ordered greedy, the extended greedy and the total-value greedy heuristic solutions are obtained for the five classes of problem instances and this is compared to the optimal solution (obtained from Martello and Toth algorithm, MTU2). The ratio between the heuristic solution value and the optimal solution value are reported. Also, the total run time taken for the execution of the algorithms excluding the time taken for input and output over a wide range of test problems are recorded. Thus, the quality, speed and the robustness of the heuristics are effectively measured.

3.3.2 Factors that Influence the Performance of the Heuristic Algorithm

The factors that affect the performance of an algorithm are the problem parameters and the test environment. The most important problem parameter is the size of the problem, *n*. The other parameters considered are the correlation between the profit (p_j) and the weight (w_j), the knapsack capacity (C) and the profit/weight ratio of the item.

The test problems randomly generated for the five problem classes are solved for 10 different *n* for the five heuristic algorithms and the exact solution algorithm. Each problem is further solved for 10 knapsack capacities in the range [100000, 1000000] with an increment of 100000. The codes are executed on small as well as large instances (largest *n* = 50000) so as to yield accurate predictions for more realistic problems. The six FORTRAN 77 codes are run on the same test problems and on the same computer configuration.

3.4 Results

The five heuristics are compared with the optimal solution algorithm to characterise their performance.

Table 3.3 compares the density-ordered greedy heuristic (H_1), the weight-ordered greedy heuristic (A) and the value-ordered greedy heuristic (B) for UKP. We see that heuristics A and B are no better than H_1 for the five problem classes. Our idea is to compare the five heuristics with the optimal solution to characterise their performance. Tables 3.4a, 3.4b, 3.5a, 3.5b, 3.6a, 3.6b, 3.7a, 3.7b, 3.8a and 3.8b summarise the results of the three heuristic algorithms H_1, H_2 and TV and the exact algorithm MTU2

(Martello and Toth, 1990) for the UKP on the generated 2500 test problems across the five problem classes.

For all problems, C ∈ [100000, 1000000] increment of 100000 for n = 50, 100, 500, 1000, 5000, 10000, 20000, 30000, 40000 and 50000.

We compare the FORTRAN 77 implementations of the following algorithms.

Code	Algorithm
DGREEDY	Density-Ordered Greedy Heuristic (H1)
WGREEDY	Weight-Ordered Greedy Heuristic (A)
VGREEDY	Value-Ordered Greedy Heuristic (B)
EXTGREED	Extended Greedy Heuristic (H2)
TOT_VAL	Total-Value Greedy Heuristic (TV)
MTU2	Martello-Toth Optimal Algorithm

The codes are provided in Appendix B.

All runs have been executed on the 200MHz Pentium Pro computer. For each data set and value of n, the tables give the average running times, expressed in seconds, computed over 5 problem instances for each of the five problem classes. Sorting and dominance check times are also separately shown.

As shown in Table 3.3, the density-ordered greedy heuristic outperforms the weight-ordered greedy heuristic (A) and the value-ordered greedy heuristic (B) in terms of the quality of solution. The performance comparison between heuristics A and B shows that B is slightly better, particularly if p and w are very strongly correlated and thereby resulting in a large number of undominated items. For problems of *Class III*, heuristic A performs better than heuristic B, possibly because of the fact that the profits (values) play insignificant role and the contribution of an item is influenced by the insertion of many (small) items. For problems with high *N*, i.e., a large number of undominated items, the value-ordered greedy heuristic (B) is better than the weight-ordered greedy heuristic (A) since there are many possible good combinations of items for assigning to the knapsack, and these are largely determined by the values of items.

Table 3.3: Comparison of solutions for H_1, A and B (for the five problem classes, 500 problem instances in each class).

Number of items, n	Uncorrelated (Class I)	Weakly Correlated (Class II)	Strongly Correlated (Class III)	Very Strongly Correlated (Class IV)	Very Very Strongly Correlated (Class V)
50	$H_1 = B > A$	$H_1 = B > A$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$[§]
100	$H_1 = B > A$	$H_1 > B > A$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
500	$H_1 > A > B$	$H_1 > B > A$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
1000	$H_1 > B > A$	$H_1 = A > B$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
5000	$H_1 > B > A$	$H_1 = A > B$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
10000	$H_1 = B > A$	$H_1 = A > B$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
20000	$H_1 = B > A$	$H_1 = A > B$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
30000	$H_1 = B > A$	$H_1 = A > B$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
40000	$H_1 = B > A$	$H_1 = A > B$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$
50000	$H_1 = B > A$	$H_1 = A > B$	$H_1 = A > B$	$H_1 > B > A$	$H_1 = B > A$

The following ten tables summarise the computational results of the five problem classes and ten problem sizes. The performance of the heuristics in terms of increase in the capacity, increase in problem size, difference in the *p/w* (density) ratio range, the running time of the heuristics in comparison to the running time of the exact solution algorithm, and the effect of dominance, is looked at.

[§] $H_1 = B > A$ means the solution by H_1 and the solution by B are equal and is better than the solution by A. Other relations are similarly defined.

The notations of the symbols used in the Tables are as below.

r = Average of R_1 (R_2, R_3) for 5 problem instances, where

$R_1 = 10^6 - (z(H_1) / z(opt)) * 10^6$; $R_2 = 10^6 - (z(H_2) / z(opt)) * 10^6$

$R_3 = 10^6 - (z(TV) / z(opt)) * 10^6$

For example, let z(opt) = 249574 ; $z(H_1)$ = 249510

then $R_1 = 10^6 - (249510/249574) * 10^6 = 10^6 - 0.999744 * 10^6 = 256$

For the *Class I* (Table 3.4a, 3.4b) and *Class II* (Table 3.5a, 3.5b) problem instances and for all the problem sizes, as the capacity of the knapsack increases, the three heuristic algorithms (H_1, H_2 and TV) give near optimal solution value and sometimes the exact solution value and take negligible amount of running time. The number of undominated items is very low thus requiring negligible running time. Since the *p*s and *w*s of problem classes I and II have no strong correlation, the difference in the density ratios does not show any clear effect in the performance of heuristics. Martello-Toth exact solution algorithm also solves the problem instances in negligible time.

For the *Class III* (Table 3.6a. 3.6b) class of UKP, density-ordered greedy and total-value greedy heuristics give near optimal solution as the capacity of the knapsack increases and give the optimal solution as the size of the problem increases. The extended greedy heuristic outperforms the other two heuristics and give the optimal solution value in almost all the problem instances *irrespective of* the size of the problem or the density range or the knapsack capacity. All the three heuristics take negligible time as there are only a few undominated items, whereas the exact solution algorithm takes some seconds (approximately 0.8 seconds) as *n* increases.

For the *Class IV* (Table 3.7a, 3.7b) class of UKP, the heuristics are far from optimal by a very small percentage. In comparison to H_1 and TV, H_2 is reasonably close to optimal. The running time for H_1 and TV are negligible but H_2 takes a few hundred seconds as the capacity of the knapsack or the problem size increases. Also, as *n* increases, the time taken to solve the problem optimally, increases. The density ratio for this class was restricted to the limit [2.0, 2.5] so as to generate problem instances that give many undominated items. Problem instances with different ratio ranges were generated and the performance of the three heuristics were studied (see Appendix C). It was found that as the density ratio values increased, total-value greedy heuristic gives the optimal solution value in neglible time whereas the exact solution algorithm used up a few seconds as the problem size increased.

For the *Class V* (Table 3.8a, 3.8b) class of UKP generated with a very large proportion of undominated items making the problem instances apparently difficult, the three heuristics give the optimal solution value for all *n* except for *n* = 100 and 500. The execution time for the heuristics is negligible when compared to the time taken to find the optimal solution value. But the time taken to solve the problems of size 100 by H_2 increases as the capacity of the knapsack increases. This is outlined in Table D1 in Appendix D where the performance ratio and the running time for all the heuristics are given. Time taken to solve the problem instances optimally increases as the capacity of the knapsack and the problem size increase. For *n* = 500 [**], the solution time is more than that for *n* = 50000.

The time taken for dominance check increases as *n* increases, which is expected; and also the number of undominated items increases as *n* increases for all the problem classes, except *Class I*. However, as mentioned earlier, the rate of growth of the number of undominated items is substantial only in *Class IV* and *Class V* problems.

[**] This is further investigated and presented in Appendix D.

Table 3.4a: Computational results for the *Uncorrelated* Class (*Class I*) of problems.
$w_j \in [10, 999]$; $p_j \in [1, 999]$ for n = 50, 100, 500 and 1000
$w_j \in [10, 4999]$; $p_j \in [1, 4999]$ for n = 5000

	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
Knapsack Capacity	† 50 2.0 0.6 0	100 1.7 0.2 0	500 2.2 0.4 0	1000 2.6 0.2 0	5000 3.4 0.2 0	50 2.0 0.6 0	100 1.7 0.2 0	500 2.2 0.4 0	1000 2.6 0.2 0	5000 3.4 0.2 0	50 2.0 0.6	100 1.7 0.2	500 2.2 0.4	1000 2.6 0.2	5000 3.4 0.2	50	100	500	1000	5000
100000	‡ 13 (0)	0 (0)	5 (0.4)	5 (0)	18 (0)	13 (0.4)	0 (0.4)	5 (0)	5 (0)	18 (0.2)	13 (0)	0 (0)	5 (0)	5 (0)	18 (0)	*1456914 (0)	2775782 (0)	7360460 (0)	7245001 (0)	32776295 (0)
200000	0 (0)	0 (0)	3 (0)	3 (0)	9 (0)	0 (0)	0 (0)	3 (0)	3 (0)	9 (0)	0 (0)	0 (0)	3 (0)	3 (0)	9 (0)	2914061 (0)	5551564 (0)	14721227 (0)	14490423 (0)	65553913 (0)
300000	0 (0)	0 (0.2)	1 (0)	1 (0.2)	5 (0)	0 (0)	0 (0)	1 (0)	1 (0.2)	5 (0)	0 (0)	0 (0)	1 (0)	1 (0)	5 (0)	4371238 (0)	8327531 (0)	22082232 (0)	21736056 (0)	98331813 (0)
400000	0 (0)	0 (0)	1 (0)	2 (0)	6 (0)	0 (0)	0 (0)	1 (0)	2 (0)	6 (0.2)	0 (0)	0 (0)	1 (0)	2 (0)	6 (0)	5828423 (0)	11103422 (0)	29443062 (0)	28981360 (0)	131109548 (0)
500000	0 (0)	0 (0.2)	1 (0.2)	1 (0)	3 (0)	0 (0)	0 (0)	1 (0)	1 (0)	3 (0)	0 (0)	0 (0)	1 (0)	1 (0)	3 (0)	7285557 (0)	13879205 (0)	36803835 (0)	36226794 (0)	163886897 (0)
600000	2 (0)	0 (0)	0 (0)	0 (0)	3 (0)	2 (0)	0 (0)	0 (0.2)	0 (0.2)	3 (0)	2 (0)	0 (0)	0 (0)	0 (0)	3 (0)	8742620 (0)	16655172 (0)	44164840 (0)	43472415 (0)	196665148 (0)
700000	0 (0)	0 (0)	0 (0)	1 (0)	2 (0)	0 (0)	0 (0)	0 (0.2)	1 (0)	2 (0.2)	0 (0)	0 (0)	0 (0)	1 (0)	2 (0)	10199823 (0)	19431023 (0)	51525671 (0)	50717731 (0)	229442890 (0)
800000	2 (0)	0 (0)	1 (0)	0 (0)	1 (0)	2 (0)	0 (0)	1 (0)	0 (0.2)	1 (0)	2 (0)	0 (0)	1 (0)	0 (0)	1 (0)	11656981 (0)	22206845 (0)	58886318 (0)	45987224 (0)	262220147 (0)
900000	0 (0)	0 (0)	1 (0.2)	0 (0.2)	2 (0)	0 (0)	0 (0)	1 (0)	0 (0)	2 (0)	0 (0)	0 (0)	1 (0)	0 (0)	2 (0)	13114264 (0)	24982812 (0)	66247269 (0)	65208645 (0)	294998098 (0)
1000000	1 (0)	0 (0)	0 (0)	1 (0)	1 (0)	1 (0)	0 (0)	0 (0)	1 (0)	1 (0)	1 (0)	0 (0)	0 (0)	1 (0)	1 (0)	14571310 (0)	27758594 (0)	73608148 (0)	72453961 (0)	327775785 (0)

† n | n = Total number of items
N | N = Number of Undominated items
t_1 | t_1 = Time taken for Dominance test (secs)
t_2 | t_2 = Time taken for Sorting (secs)

‡ r | r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value
(t) | t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z | z = optimal solution value
(t) | t = time in seconds

$R_1 = 10^6-(z(H_1)/z(opt))*10^6$, $R_2 = 10^6-(z(H_2)/z(opt))*10^6$, $R_3 = 10^6-(z(TV)/z(opt))*10^6$

Table 3.4b: Computational results for the *Uncorrelated* Class (*Class I*) of problems.
$w_j \in [10, 9999]$; $p_j \in [1, 9999]$

	Relative Performance of Heuristic Algorithms																			
Knapsack Capacity	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	†10000 2.6 0.2 0	20000 2.8 0.4 0	30000 2.8 0.6 0	40000 3.2 0.2 0	50000 1.8 0.6 0	10000 2.6 0.2 0	20000 2.8 0.4 0	30000 2.8 0.6 0	40000 3.2 0.2 0	50000 1.8 0.6 0	10000 2.6 0.2	20000 2.8 0.4	30000 2.8 0.6	40000 3.2 0.2	50000 1.8 0.6	10000	20000	30000	40000	50000
100000	‡ 0 (0.2)	2 (0.2)	2 (0)	3 (0.2)	0 (0.4)	0 (0.2)	2 (0)	2 (0.2)	3 (0.2)	0 (0.4)	0 (0)	2 (0.2)	2 (0)	3 (0)	0 (0)	*68687912 (0)	77491311 (0.4)	86106715 (0)	84496956 (0.8)	91060858 (0.2)
200000	2 (0)	13 (0)	15 (0)	3 (0)	15 (0.2)	2 (0)	13 (0)	15 (0)	3 (0)	15 (0)	2 (0)	13 (0)	15 (0)	3 (0)	15 (0)	137377012 (0)	154984788 (0)	172214132 (0)	168994018 (0)	182122346 (0)
300000	3 (0)	8 (0)	5 (0)	0 (0)	3 (0)	3 (0)	8 (0)	5 (0)	0 (0)	3 (0)	3 (0)	8 (0)	5 (0)	0 (0)	3 (0)	206066517 (0)	232477685 (0)	258321549 (0)	253492000 (0)	273184567 (0)
400000	2 (0)	5 (0)	3 (0)	1 (0)	0 (0)	2 (0)	5 (0)	3 (0)	1 (0)	0 (0)	2 (0)	5 (0)	3 (0)	1 (0)	0 (0)	274756995 (0)	309971627 (0)	344428264 (0)	337988956 (0)	364245514 (0)
500000	0 (0)	1 (0)	4 (0)	2 (0)	2 (0)	0 (0)	1 (0)	4 (0)	2 (0)	2 (0)	0 (0)	1 (0)	4 (0)	2 (0)	2 (0)	357674818 (0)	387465085 (0)	430535681 (0)	422486018 (0)	455307961 (0)
600000	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	412136388 (0)	464959052 (0)	516643098 (0)	524790000 (0.2)	546371160 (0)
700000	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	480826866 (0)	542453442 (0)	602749812 (0)	591480956 (0)	637432997 (0)
800000	0 (0)	0 (0)	2 (0)	1 (0)	0 (0)	0 (0)	0 (0)	2 (0)	1 (0)	0 (0)	0 (0)	0 (0)	2 (0)	1 (0)	0 (0)	549515799 (0)	619947073 (0)	688857265 (0)	675978018 (0)	728495465 (0)
900000	0 (0)	0 (0.2)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	618206225 (0)	697441464 (0)	774964910 (0)	760476000 (0)	819558664 (0)
1000000	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	686896028 (0)	774935095 (0)	861071787 (0)	844972956 (0)	910620501 (0)

† n / N / t_1 / t_2

n = Total number of items
N = Number of Undominated items
t_1 = Time taken for Dominance test (secs)
t_2 = Time taken for Sorting (secs)

‡ r / (t)

r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value
t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z / (t)

z = optimal solution value
t = time in seconds

$R_1 = 10^6-(z(H_1) / z(opt))*10^6$, $R_2 = 10^6-(z(H_2) / z(opt))*10^6$, $R_3 = 10^6-(z(TV) / z(opt))*10^6$

Table 3.5a: Computational results for the *Weakly Correlated* Class (*Class II*) of problems.

$w_j \in [10, 999]$; $p_j \in [w_j - 100, w_j + 100]$

Knapsack Capacity	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	† 50 1.8 0 0	100 2.1 0.2 0	500 4.6 0.6 0	1000 2.8 0.2 0	5000 3.4 0.8 0	50 1.8 0 0	100 2.1 0.2 0	500 4.6 0.6 0	1000 2.8 0.2 0	5000 3.4 0.8 0	50 1.8 0	100 2.1 0.2	500 4.6 0.6	1000 2.8 0.2	5000 3.4 0.8	50	100	500	1000	5000
100000	‡ 0 (0)	0 (0.2)	5 (0)	6 (0)	0 (0.4)	0 (0)	0 (0)	1 (0)	5 (0)	0 (0)	0 (0)	0 (0)	3 (0)	6 (0.2)	0 (0)	*474806 (0)	555592 (0)	624331 (0)	790218 (0)	899341 (0.2)
200000	6 (0)	0 (0.2)	1 (0.4)	1 (0)	12 (0)	6 (0)	0 (0)	0 (0)	1 (0)	12 (0)	6 (0)	0 (0)	1 (0)	1 (0)	12 (0)	949615 (0)	1111224 (0)	1248694 (0)	1580461 (0)	1798672 (0)
300000	0 (0)	0 (0)	0 (0)	1 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6 (0)	0 (0)	0 (0)	0 (0)	1 (0)	6 (0)	1424451 (0)	1666856 (0)	1873057 (0.2)	2370716 (0)	2698012 (0)
400000	0 (0)	0 (0)	1 (0)	1 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3 (0)	0 (0)	0 (0)	1 (0)	1 (0)	3 (0)	1899275 (0)	2222480 (0)	2497396 (0)	3160969 (0)	3597368 (0)
500000	2 (0)	0 (0)	0 (0)	1 (0.2)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	2 (0)	2 (0)	0 (0)	0 (0)	1 (0)	2 (0)	2374084 (0)	2778112 (0)	3121760 (0)	3951217 (0)	4496718 (0)
600000	0 (0)	0 (0)	0 (0)	0 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2 (0)	2848931 (0)	3333744 (0)	3746122 (0)	4741475 (0)	5396083 (0)
700000	0 (0)	0 (0)	1 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	1 (0)	0 (0)	1 (0)	3323760 (0)	3889376 (0)	4370462 (0)	5531726 (0)	6295440 (0)
800000	1 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	1 (0)	3798569 (0)	4390996 (0)	4994825 (0)	6321956 (0)	7194790 (0)
900000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4273399 (0)	5000628 (0)	5619188 (0)	7112212 (0.2)	8094147 (0)
1000000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4748229 (0)	5556260 (0)	6243528 (0)	7902462 (0)	8993491 (0)

† n
N
t_1
t_2

n = Total number of items

N = Number of Undominated items

t_1 = Time taken for Dominance test (secs)

t_2 = Time taken for Sorting (secs)

‡ r
(t)

r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value

t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z
(t)

z = optimal solution value

t = time in seconds

$R_1 = 10^6-(z(H_1) / z(opt))*10^6$, $R_2 = 10^6-(z(H_2) / z(opt))*10^6$, $R_3 = 10^6-(z(TV) / z(opt))*10^6$

Table 3.5b: Computational results for the *Weakly Correlated* Class (*Class II*) of problems.
$w_j \in [10, 9999]$; $p_j \in [w_j - 100, w_j + 100]$

Knapsack Capacity	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	†10000 2.6 0 0	20000 4.6 0.2 0	30000 4.2 0.6 0	40000 3.6 0.2 0	50000 3.4 0.4 0	10000 2.6 0 0	20000 4.6 0.2 0	30000 4.2 0.6 0	40000 3.6 0.2 0	50000 3.4 0.4 0	10000 2.6 0	20000 4.6 0.2	30000 4.2 0.6	40000 3.6 0.2	50000 3.4 0.4	10000	20000	30000	40000	50000
100000	‡ 1 (0.4)	0 (0.2)	5 (0.2)	7 (0)	5 (0.2)	1 (0)	0 (0)	5 (0.2)	7 (0)	5 (0.2)	1 (0.2)	0 (0)	5 (0)	7 (0)	5 (0)	*731141 (0)	789126 (0)	931337 (0)	979542 (0)	1018664 (0)
200000	9 (0)	6 (0)	11 (0)	14 (0)	7 (0)	9 (0.2)	6 (0)	11 (0.2)	14 (0)	7 (0)	9 (0)	6 (0)	11 (0)	14 (0)	7 (0)	1696296 (0)	1578253 (0)	1862696 (0)	1959085 (0)	2037332 (0)
300000	0 (0)	1 (0)	3 (0.2)	2 (0)	0 (0)	0 (0)	1 (0)	3 (0)	2 (0)	0 (0)	0 (0)	1 (0)	3 (0)	2 (0)	0 (0)	2544456 (0)	2367383 (0)	2794069 (0)	2938628 (0)	3056000 (0.2)
400000	0 (0)	0 (0.2)	3 (0)	3 (0)	1 (0)	0 (0)	0 (0)	3 (0)	3 (0)	1 (0)	0 (0)	0 (0)	3 (0)	3 (0)	1 (0)	3392604 (0)	3156515 (0)	3725447 (0)	3918172 (0.2)	4074664 (0.2)
500000	4 (0)	2 (0)	2 (0)	4 (0)	3 (0)	4 (0)	2 (0)	2 (0)	4 (0)	3 (0)	4 (0)	2 (0)	2 (0)	4 (0)	3 (0)	4240760 (0)	3945649 (0)	4656818 (0)	4897719 (0.2)	5093332 (0.2)
600000	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0.2)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	5088919 (0)	4734783 (0)	5588195 (0)	5877266 (0.2)	6112000 (0)
700000	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	5937067 (0)	5523916 (0)	6519572 (0)	6856805 (0.2)	7130664 (0.2)
800000	1 (0)	2 (0)	1 (0)	2 (0)	0 (0.2)	1 (0)	2 (0.2)	1 (0)	2 (0)	0 (0)	1 (0)	2 (0)	1 (0)	2 (0)	0 (0)	6785216 (0)	6313043 (0)	7450943 (0.6)	7836359 (0)	8149332 (0)
900000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0.2)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7633383 (0)	7102171 (0)	8383174 (0)	8815906 (0)	9168000 (0)
1000000	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0.2)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	8481530 (0)	7891299 (0)	9313700 (0.2)	9795452 (0)	10186664 (0)

† n / N / t_1 / t_2

n = Total number of items
N = Number of Undominated items
t_1 = Time taken for Dominance test (secs)
t_2 = Time taken for Sorting (secs)

‡ r / (t)

r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value
t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z / (t)

z = optimal solution value
t = time in seconds

$R_1 = 10^6-(z(H_1)/z(opt))*10^6$, $R_2 = 10^6-(z(H_2)/z(opt))*10^6$, $R_3 = 10^6-(z(TV)/z(opt))*10^6$

Table 3.6a: Computational results for the *Strongly Correlated* Class (*Class III*) of problems.
$w_j \in [10, 999]$; $p_j = w_j + 100$ for n = 50, 100, 500 and 1000
$w_j \in [10, 4999]$; $p_j = w_j + 100$ for n = 5000

	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
Knapsack Capacity	R_1					R_2					R_3					z				
	† 50 2.4 0 0	100 2.9 0 0	500 5.8 0.2 0	1000 7.2 0.6 0	5000 8.6 0.2 0	50 2.4 0 0	100 2.9 0 0	500 5.8 0.2 0	1000 7.2 0.6 0	5000 8.6 0.2 0	50 2.4 0	100 2.9 0	500 5.8 0.2	1000 7.2 0.6	5000 8.6 0.2	50	100	500	1000	5000
100000	‡ 50 (0.2)	19 (0)	4 (0.2)	4 (0)	3 (0)	50 (0)	0 (0)	0 (0)	0 (0.2)	0 (0.2)	50 (0)	19 (0)	4 (0)	4 (0)	3 (0.2)	*638338 (0)	768360 (0)	1002300 (0)	1063600 (0)	1035640 (0)
200000	22 (0.2)	7 (0)	3 (0)	2 (0)	5 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	22 (0)	7 (0)	3 (0)	2 (0)	5 (0)	1276718 (0)	1536759 (0)	2004620 (0)	2127240 (0)	2071300 (0)
300000	7 (0)	2 (0.2)	1 (0)	1 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7 (0)	2 (0)	1 (0)	1 (0)	4 (0)	1915116 (0)	2305180 (0)	3006960 (0)	3190880 (0)	3106960 (0)
400000	5 (0)	5 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5 (0)	5 (0)	1 (0)	1 (0)	1 (0)	2553497 (0)	3073580 (0)	4009300 (0)	4254520 (0)	4142640 (0)
500000	4 (0)	3 (0)	1 (0)	1 (0)	2 (0.2)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (0)	3 (0)	1 (0)	0 (0)	1 (0)	3191878 (0)	3841980 (0)	5011620 (0)	5318160 (0)	5178300 (0)
600000	0 (0)	1 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	1 (0)	3830277 (0)	4610399 (0)	5182140 (0)	6381800 (0)	6213960 (0)
700000	0 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3 (0)	0 (0)	0 (0)	0 (0)	4468658 (0)	5378799 (0)	7016300 (0)	7445440 (0)	7249640 (0)
800000	6 (0)	2 (0)	1 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6 (0)	2 (0)	0 (0)	0 (0)	1 (0)	5107017 (0)	6147179 (0)	8018620 (0)	8509080 (0)	8285300 (0)
900000	5 (0)	1 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5 (0)	1 (0)	0 (0)	0 (0)	1 (0)	5745418 (0)	6915599 (0)	9020960 (0)	9572720 (0.2)	9320960 (0)
1000000	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6383799 (0)	7684020 (0)	10023300 (0)	10636360 (0)	10356640 (0)

† n
N
t_1
t_2

n = Total number of items
N = Number of Undominated items
t_1 = Time taken for Dominance test (secs)
t_2 = Time taken for Sorting (secs)

‡ r
(t)

r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value
t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z
(t)

z = optimal solution value
t = time in seconds

$R_1 = 10^6-(z(H_1) / z(opt))*10^6$, $R_2 = 10^6-(z(H_2) / z(opt))*10^6$, $R_3 = 10^6-(z(TV) / z(opt))*10^6$

Table 3.6b: Computational results for the *Strongly Correlated* Class (*Class III*) of problems.

$w_j \in [10, 9999]$; $p_j = w_j + 100$

Knapsack Capacity	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	†10000	20000	30000	40000	50000	10000	20000	30000	40000	50000	10000	20000	30000	40000	50000	10000	20000	30000	40000	50000
	6.0 0.4 0	5.0 0 0	6.0 0 0	6.2 0.4 0	6.6 0.2 0	6.0 0.4 0	5.0 0 0	6.0 0 0	6.2 0.4 0	6.6 0.2 0	6.0 0.4	5.0 0	6.0 0	6.2 0.4	6.6 0.2					
100000	‡ 2 (0)	4 (0.2)	4 (0)	2 (0)	1 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	2 (0)	4 (0)	4 (0)	2 (0)	1 (0)	*1081800 (0.2)	1015120 (0.4)	1063600 (0)	1033320 (0.6)	1020499 (0.8)
200000	1 (0.2)	3 (0)	2 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	3 (0)	2 (0)	2 (0)	2 (0)	2163620 (0.2)	2030260 (0.2)	2127240 (0.4)	2066640 (0.2)	2041000 (0)
300000	1 (0)	1 (0)	1 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	1 (0)	0 (0)	1 (0)	3245440 (0)	3045440 (0.2)	3190880 (0.2)	3100000 (0.4)	3061520 (0.2)
400000	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	4327260 (0.2)	4060580 (0)	4254520 (0)	4133320 (0.2)	4082039 (0.6)
500000	0 (0)	1 (0)	1 (0.2)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	1 (0)	1 (0)	5409080 (0)	5075720 (0.6)	5318160 (0.2)	5166640 (0.8)	5102540 (0.4)
600000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6490900 (0)	6090899 (0)	6381800 (0.2)	6200000 (0.2)	6123060 (0)
700000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7572720 ((0)	7106039 (0)	7445440 (0)	7233320 (0)	7143579 (0.8)
800000	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	8654540 (0.2)	8121179 (0)	8509080 (0.6)	8266640 (0.4)	8164080 (0.4)
900000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	9736360 (0)	9136360 (0.4)	9572720 (0)	9300000 (0.4)	9184600 (0.4)
1000000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	10818180 (0)	10151500 (0.2)	10636360 (0.2)	10333320 (0.4)	10205119 (0.4)

†	
n	n = Total number of items
N	N = Number of Undominated items
t_1	t_1 = Time taken for Dominance test (secs)
t_2	t_2 = Time taken for Sorting (secs)

‡ r / (t): r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value

t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z / (t): z = optimal solution value

t = time in seconds

$R_1 = 10^6-(z(H_1) / z(opt))*10^6$, $R_2 = 10^6-(z(H_2) / z(opt))*10^6$, $R_3 = 10^6-(z(TV) / z(opt))*10^6$

Table 3.7a: Computational results for the *Very Strongly Correlated* Class (*Class IV*) of problems.
$w_j \in [1, 999]$; $p_j \in [1, 999]$ and $2.0 \le p_j / w_j \le 2.5$ for n = 50, 100, 500 and 1000
$w_j \in [1, 9999]$; $p_j \in [1, 9999]$ and $2.0 \le p_j / w_j \le 2.5$ for n = 5000

Knapsack Capacity	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	† 50 18 0.2 0	100 26 0 0	500 49.8 0 0	1000 83.4 1 0	5000 173.4 0.8 0	50 18 0.2 0	100 26 0 0	500 49.8 0 0	1000 83.4 1 0	5000 173.4 0.8 0	50 18 0.2	100 26 0	500 49.8 0	1000 83.4 1	5000 173.4 0.8	50	100	500	1000	5000
100000	‡ 169 (0)	217 (0)	66 (0)	516 (0)	338 (0)	120 (0.2)	29 (0.4)	29 (1.8)	357 (0.4)	182 (0)	586 (0)	411 (0)	62 (0)	2129 (0)	442 (0)	*248790 (0)	249443 (0)	249998 (0.4)	249926 (0.2)	249987 (0.2)
200000	88 (0)	140 (0)	57 (0.2)	386 (0.2)	207 (0)	95 (1.0)	41 (1.8)	16 (5.4)	251 (0.2)	41 (0)	47 (0)	9 (0)	162 (0)	747 (0)	431 (0)	497600 (0)	498898 (0)	499999 (0)	499875 (0.2)	499975 (0.2)
300000	33 (0)	99 (0)	42 (0)	217 (0)	181 (0)	52 (3.0)	32 (3.8)	8 (12.0)	137 (0)	16 (0.4)	11 (0)	75 (0)	39 (0)	528 (0)	294 (0)	746410 (0)	748348 (0)	749998 (0)	749820 (0.4)	749964 (0)
400000	45 (0)	86 (0)	24 (0)	192 (0)	123 (0)	30 (4.8)	15 (6.6)	2 (23.0)	74 (0.2)	21 (0.2)	37 (0)	24 (0)	17 (0)	358 (0)	261 (0)	995215 (0)	997802 (0)	999998 (0)	999766 (0.2)	999952 (0.4)
500000	18 (0)	99 (0)	21 (0)	261 (0)	82 (0)	21 (8.4)	17 (11.4)	2 (34.6)	168 (0)	23 (0.2)	12 (0)	28 (0)	13 (0)	995 (0)	117 (0)	1244031 (0)	1247252 (0)	1249999 (0)	1249718 (0)	1249941 (0)
600000	32 (0)	87 (0)	7 (0)	156 (0)	120 (0)	42 (11.6)	19 (16.2)	7 (50.2)	91 (0.2)	16 (0.8)	16 (0)	20 (0)	5 (0)	211 (0)	153 (0)	1492838 (0)	1496706 (0)	1499998 (0)	1499668 (0)	1499928 (0)
700000	24 (0)	83 (0)	8 (0)	105 (0)	121 (0)	10 (14.8)	9 (20.8)	4 (71.0)	60 (0.2)	13 (1.0)	15 (0)	80 (0)	6 (0)	475 (0)	168 (0)	1741647 (0)	1746156 (0)	1749998 (0)	1749615 (0)	1749917 (0.2)
800000	19 (0)	73 (0)	13 (0)	59 (0)	127 (0)	9 (20.4)	2 (27.8)	2 (86.2)	60 (0.6)	12 (1.4)	12 (0)	17 (0)	3 (0)	472 (0)	143 (0)	1990456 (0.2)	1995611 (0)	1999998 (0)	1999563 (0)	1999905 (0.2)
900000	13 (0)	83 (0)	11 (0.2)	102 (0)	75 (0.2)	10 (23.0)	7 (37.4)	1 (111.2)	98 (0.6)	9 (1.6)	9 (0)	21 (0)	8 (0)	420 (0)	171 (0)	2239268 (0)	2245060 (0))	2249998 (0)	2249509 (0.2)	2249894 (0)
1000000	14 (0)	64 (0)	9 (0)	41 (0)	82 (0)	7 (30.2)	11 (44.4)	2 (136.2)	72 (0.6)	9 (1.8)	12 (0)	12 (0)	7 (0)	232 (0)	78 (0)	2488073 (0)	2494512 (0)	2499999 (0)	2499459 (0)	2499882 (0.2)

† n / N / t_1 / t_2
n = Total number of items
N = Number of Undominated items
t_1 = Time taken for Dominance test (secs)
t_2 = Time taken for Sorting (secs)

‡ r / (t)
r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value
t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z / (t)
z = optimal solution value
t = time in seconds

$R_1 = 10^6 - (z(H_1) / z(opt))*10^6$, $R_2 = 10^6 - (z(H_2) / z(opt))*10^6$, $R_3 = 10^6 - (z(TV) / z(opt))*10^6$

Table 3.7b: Computational results for the *Very Strongly Correlated* Class (*Class IV*) of problems.
$w_j \in [1, 9999]$; $p_j \in [1, 9999]$ and $2.0 \le p_j / w_j \le 2.5$

	Relative Performance of Heuristic Algorithms																			
Knapsack Capacity	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	† 10000 241.2 1.8 0	20000 285.2 3.4 0	30000 377.6 6.6 0	40000 399.6 10 0.2	50000 401.8 19 0.2	10000 241.2 1.8 0	20000 285.2 3.4 0	30000 377.6 6.6 0	40000 399.6 10 0.2	50000 401.8 19 0.2	10000 241.2 1.8	20000 285.2 3.4	30000 377.6 6.6	40000 399.6 10	50000 401.8 19	10000	20000	30000	40000	50000
100000	‡ 210 (0)	72 (0)	102 (0)	114 (0)	80 (0.4)	183 (0.4)	44 (1.2)	0 (0.48)	0 (3.0)	12 (11.8)	255 (0)	400 (0)	121 (0)	138 (0)	104 (0)	*249999 (0.2)	250000 (0.2)	250000 (0.24)	250000 (0.8)	250000 (0.8)
200000	162 (0)	24 (0)	16 (0)	50 (0)	29 (0)	78 (0.6)	15 (4.8)	0 (1.68)	0 (12.8)	8 (50.2)	233 (0)	158 (0)	147 (0)	39 (0)	104 (0)	499999 (0)	500000 (0.2)	500000 (0.36)	500000 (0.4)	500000 (1.0)
300000	149 (0)	25 (0)	30 (0.12)	30 (0)	5 (0)	58 (1.2)	8 (11.0)	0 (4.08)	0 (30.2)	0 (120.8)	231 (0)	193 (0)	57 (0)	18 (0)	8 (0)	750000 (0.2)	750000 (0.4)	750000 (0.12)	750000 (0.8)	750000 (0.8)
400000	131 (0)	29 (0)	16 (0.12)	28 (0)	14 (0)	33 (2.2)	15 (20.8)	0 (7.08)	0 (53.8)	2 (211.4)	233 (0)	119 (0)	68 (0)	60 (0)	25 (0)	999999 (0.2)	1000000 (0.2)	1000000 (0.12)	1000000 (0.4)	1000000 (1.2)
500000	100 (0)	14 (0)	14 (0)	19 (0)	11 (0)	52 (3.8)	8 (28.4)	0 (12.0)	0 (89.2)	3 (322.4)	103 (0)	71 (0)	13 (0)	19 (0)	6 (0)	1249999 (0.6)	1250000 (0.2)	1250000 (0.36)	1250000 (0.4)	1250000 (0.6)
600000	103 (0.2)	12 (0)	12 (0)	8 (0)	4 (0)	31 (5.4)	5 (46.0)	0 (21.6)	0 (126.8)	0 (467.4)	83 (0)	40 (0)	41 (0)	7 (0)	7 (0)	1499999 (0.2)	1500000 (0.4)	1500000 (0.12)	1500000 (0.6)	1500000 (0.8)
700000	100 (0)	9 (0)	10 (0)	14 (0)	3 (0)	33 (7.6)	2 (61.6)	0 (38.8)	0 (159.2)	1 (599.4)	77 (0)	57 (0)	36 (0)	8 (0)	15 (0)	1749999 (0.2)	1750000 (0.2)	1750000 (0)	1750000 (0.4)	1750000 (1.0)
800000	88 (0.2)	14 (0)	6 (0)	18 (0)	10 (0)	44 (9.2)	0 (81.4)	0 (49.2)	0 (215.2)	2 (791.2)	74 (0)	101 (0)	9 (0)	9 (0)	25 (0)	1999999 (0.4)	2000000 (0)	2000000 (0)	2000000 (0.8)	2000000 (0.8)
900000	84 (0)	3 (0)	10 (0)	11 (0)	8 (0)	33 (11.4)	4 (100.2)	0 (63.2)	0 (276.8)	0 (999.2)	37 (0)	3 (0)	8 (0)	5 (0)	2 (0)	2249999 (0.2)	2250000 (0.4)	2250000 (0.24)	2250000 (0.4)	2250000 (0.8)
1000000	81 (0)	6 (0)	10 (0)	90 (0)	3 (0)	33 (14.2)	28 (123.6)	0 (77.6)	0 (359.0)	1 (1236.6)	46 (0)	53 (0)	7 (0)	7 (0)	6 (0)	2499999 (0.4)	2500000 (0.2)	2500000 (0.36)	2500000 (0.6)	2500000 (0.6)

† n
N
t_1
t_2

- n = Total number of items
- N = Number of Undominated items
- t_1 = Time taken for Dominance test (secs)
- t_2 = Time taken for Sorting (secs)

‡ r
(t)

- r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded, the smaller the value of R, the closer the heuristic is to the optimal solution value
- t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z
(t)

- z = optimal solution value
- t = time in seconds

$R_1 = 10^6 - (z(H_1) / z(opt))*10^6$, $R_2 = 10^6 - (z(H_2) / z(opt))*10^6$, $R_3 = 10^6 - (z(TV) / z(opt))*10^6$

Table 3.8a: Computational results for the *Very Very Strongly Correlated* Class (*Class V*) of problems.

$w_j \in [1, 999]$; $p_j = w_j * [(w_j - \min w_j + 1) / (\max w_j - \min w_j + 1)] * 100$ for n = 50, 100, 500 and 1000

$w_j \in [1, 99999]$; $p_j = w_j * [(w_j - \min w_j + 1) / (\max w_j - \min w_j + 1)] * 100$ for n = 5000

Knapsack Capacity	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	† 50 37.8 0 0	100 97 0.12 0	500 399 0.4 0	1000 954.8 2.6 1	5000 4882.2 54.2 25.4	50 37.8 0 0	100 97 0.12 0	500 399 0.4 0	1000 954.8 2.6 1	5000 4882.2 54.2 25.4	50 37.8 0	100 97 0.12	500 399 0.4	1000 954.8 2.6	5000 4882.2 54.2	50	100	500	1000	5000
100000	‡ 0 (0)	14 (0)	1 (0.4)	0 (0)	0 (0)	0 (1.44)	14 (0)	1 (0.2)	0 (0)	0 (0)	0 (0)	14 (0)	430 (0)	0 (0)	0 (0.12)	*9695154 (0)	9913484 (0.24)	9931558 (5244.8)	9982693 (0.48)	9996341 (5.16)
200000	0 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (6.24)	3 (0)	0 (0)	0 (0)	0 (0.12)	0 (0)	4 (0)	0 (0)	0 (0)	0 (0.12)	19392312 (0)	19854421 (0.24)	19873372 (21009.4)	19965607 (0.6)	19992682 (11.52)
300000	0 (0)	24 (0)	5 (0)	0 (0)	0 (0)	0 (14.28)	17 (0.12)	5 (0)	0 (0)	0 (0)	0 (0)	24 (0)	5 (0)	0 (0)	0 (0)	29089036 (0)	29790819 (0.12)	29820664 (4945)	29948762 (0.96)	29989025 (18.72)
400000	0 (0)	0 (0)	0 (0.2)	0 (0)	0 (0)	0 (24.48)	46 (0.12)	0 (0)	0 (0)	0 (0)	0 (0)	203 (0)	202 (0)	0 (0)	0 (0)	38785235 (0)	39728546 (0)	39768587 (200131.6)	39932125 (1.08)	39985370 (9.24)
500000	0 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (57.96)	164 (0.12)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0.12)	48482934 (0)	49662860 (0)	49726376 (2.8)	49915670 (1.56)	49981712 (12.24)
600000	0 (0)	11 (0)	0 (0.2)	0 (0)	0 (0)	0 (80.76)	91 (0.48)	0 (0)	0 (0.12)	0 (0)	0 (0)	11 (0)	0 (0)	0 (0)	0 (0.24)	58178711 (0)	59602018 (0.12)	59661189 (73.6)	59899459 (1.2)	59978056 (15.36)
700000	0 (0)	5 (0)	0 (0.2)	0 (0)	0 (0.24)	0 (121.92)	0 (0.48)	0 (0)	0 (0)	0 (0)	0 (0)	5 (0)	0 (0)	0 (0)	0 (0)	67875874 (0)	69534576 (0.24)	69601180 (24520.6)	69883503 (1.32)	69974407 (22.92)
800000	0 (0)	0 (0)	1 (0)	0 (0.12)	0 (0)	0 (161.28)	90 (0.72)	1 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	0 (0)	0 (0)	77572736 (0)	79467436 (0.24)	79551418 (4483.4)	79867703 (1.08)	79970760 (22.44)
900000	0 (0)	5 (0.12)	0 (0)	0 (0.12)	0 (0)	0 (205.92)	87 (0.84)	0 (0)	0 (0)	0 (0.24)	0 (0)	5 (0)	0 (0)	0 (0)	0 (0.12)	87269189 (0)	89404279 (0)	89497659 (4150.2)	89852196 (1.32)	89967106 (20.28)
1000000	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (264)	0 (1.32)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0.12)	96966602 (0)	99352065 (0)	99453488 (355.6)	99836679 (1.2)	99963456 (21.84)

†	
n	n = Total number of items
N	N = Number of Undominated items
t_1	t_1 = Time taken for Dominance test (secs)
t_2	t_2 = Time taken for Sorting (secs)

‡ r / (t): r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value

t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z / (t): z = optimal solution value

t = time in seconds

$R_1 = 10^6-(z(H_1) / z(opt))*10^6$, $R_2 = 10^6-(z(H_2) / z(opt))*10^6$, $R_3 = 10^6-(z(TV) / z(opt))*10^6$

Table 3.8b: Computational results for the *Very Very Strongly Correlated* Class (*Class V*) of problems.
$w_j \in [1, 99999]$; $p_j = w_j * [(w_j - \min w_j + 1) / (\max w_j - \min w_j + 1)] * 100$

Knapsack Capacity	Relative Performance of Heuristic Algorithms																			
	Density-ordered Greedy, H_1					Extended Greedy, H_2					Total-value Greedy, TV					Optimal Solution, MTU2				
	R_1					R_2					R_3					z				
	†10000 9491.4 201.4 102.4	20000 18123 834.4 378.6	30000 25869.2 1911 764	40000 32908 3293 1229.4	50000 39322.2 8796.6 3794	10000 9491.4 201.4 102.4	20000 18123 834.4 378.6	30000 25869.2 1911 764	40000 32908 3293 1229.4	50000 39322.2 8796.6 3794	10000 9491.4 201.4	20000 18123 834.4	30000 25869.2 1911	40000 32908 3293	50000 39322.2 8796.6	10000	20000	30000	40000	50000
100000	‡ 0 (0)	0 (0.24)	0 (0)	0 (0.12)	0 (0.36)	0 (0)	0 (0.24)	0 (0.24)	0 (0.12)	0 (0.36)	0 (0)	0 (0)	0 (0.12)	0 (0.12)	0 (0.12)	*9997420 (0)	9998220 (0.12)	9998940 (0.72)	9999420 (0.84)	9999500 (0.96)
200000	0 (0)	0 (0.12)	0 (0.12)	0 (0.12)	0 (0.24)	0 (0)	0 (0)	0 (0.12)	0 (0)	0 (0.12)	0 (0)	0 (0)	0 (0)	0 (0.48)	0 (0.48)	19994841 (13.8)	19996440 (48.48)	19997880 (0.48)	19998840 (0.6)	19999000 (1.08)
300000	0 (0)	0 (0)	0 (0.12)	0 (0)	0 (0)	0 (0.12)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0.12)	0 (0)	0 (0.36)	0 (0.36)	29992262 (34.44)	29994661 (39.96)	29996820 (0.72)	29998260 (0.72)	29998500 (1.08)
400000	0 (0.24)	0 (0)	0 (0.12)	0 (0.24)	0 (0)	0 (0.12)	0 (0.12)	0 (0)	0 (0.24)	0 (0.12)	0 (0.12)	0 (0)	0 (0.12)	0 (0.36)	0 (0.48)	39989684 (34.08)	39992881 (98.52)	39995760 (148.2)	39997680 (0.48)	39998000 (1.32)
500000	0 (0)	0 (0)	0 (0.12)	0 (0.12)	0 (0.24)	0 (0)	0 (0.12)	0 (0.12)	0 (0)	0 (0)	0 (0)	0 (0.12)	0 (0.24)	0 (0.36)	0 (0.48)	49987106 (35.28)	49991102 (92.88)	49994701 (310.2)	49997100 (0.84)	49997500 (1.2)
600000	0 (0)	0 (0.12)	0 (0.12)	0 (0.24)	0 (0.24)	0 (0)	0 (0)	0 (0)	0 (0.24)	0 (0.24)	0 (0)	0 (0)	0 (0)	0 (0.36)	0 (0.24)	59984529 (34.44)	59989324 (210.72)	59993641 (550.2)	59996520 (0.48)	59997000 (0.84)
700000	0 (0)	0 (0.12)	0 (0)	0 (0)	0 (0)	0 (0.12)	0 (0)	0 (0.24)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0.36)	0 (0.36)	69981952 (33.24)	69987545 (201)	69992581 (439.8)	69995940 (0.96)	69996500 (1.2)
800000	0 (0.12)	0 (0)	0 (0.12)	0 (0)	0 (0.36)	0 (0)	0 (0.24)	0 (0)	0 (0)	0 (0.12)	0 (0.12)	0 (0.48)	0 (0.24)	0 (0.24)	0 (0.36)	79979375 (32.16)	79985766 (215.4)	79991522 (441.6)	79995360 (0.6)	79996000 (1.08)
900000	0 (0.12)	0 (0.12)	0 (0.12)	0 (0.24)	0 (0)	0 (0)	0 (0.24)	0 (0.12)	0 (0)	0 (0.24)	0 (0.12)	0 (0)	0 (0.12)	0 (0.24)	0 (0.36)	89976800 (39.36)	89983988 (207.24)	89990463 (391.8)	89994781 (499.8)	89995500 (339.24)
1000000	0 (0.24)	0 (0.24)	0 (0)	0 (0.24)	0 (0.36)	0 (0.12)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0.12)	0 (0.36)	0 (0)	99974224 (48.6)	99982210 (244.68)	99989403 (381)	99994201 (425.4)	99995000 (362.28)

† n
N
t_1
t_2

n = Total number of items
N = Number of Undominated items
t_1 = Time taken for Dominance test (secs)
t_2 = Time taken for Sorting (secs)

‡ r
(t)

r = Average of R_1, R_2 or R_3 for 5 problem instances; the values of R are coded,the smaller the value of R, the closer the heuristic is to the optimal solution value
t = total CPU - seconds, excluding sorting, dominance test and data input and output

* z
(t)

z = optimal solution value
t = time in seconds

$R_1 = 10^6 - (z(H_1) / z(opt))*10^6$, $R_2 = 10^6 - (z(H_2) / z(opt))*10^6$, $R_3 = 10^6 - (z(TV) / z(opt))*10^6$

3.5 Analysis of the Results

This section focuses on the analysis based on computational experience and comparison of heuristics.

The figures 3.2 to 3.13 show the performance of the three heuristics for *Class IV* and *Class V* problem instances for some small and large problem sizes.

For the first three classes *Class I, Class II* and *Class III* all the heuristics are either close to optimal or give the optimal solution value.

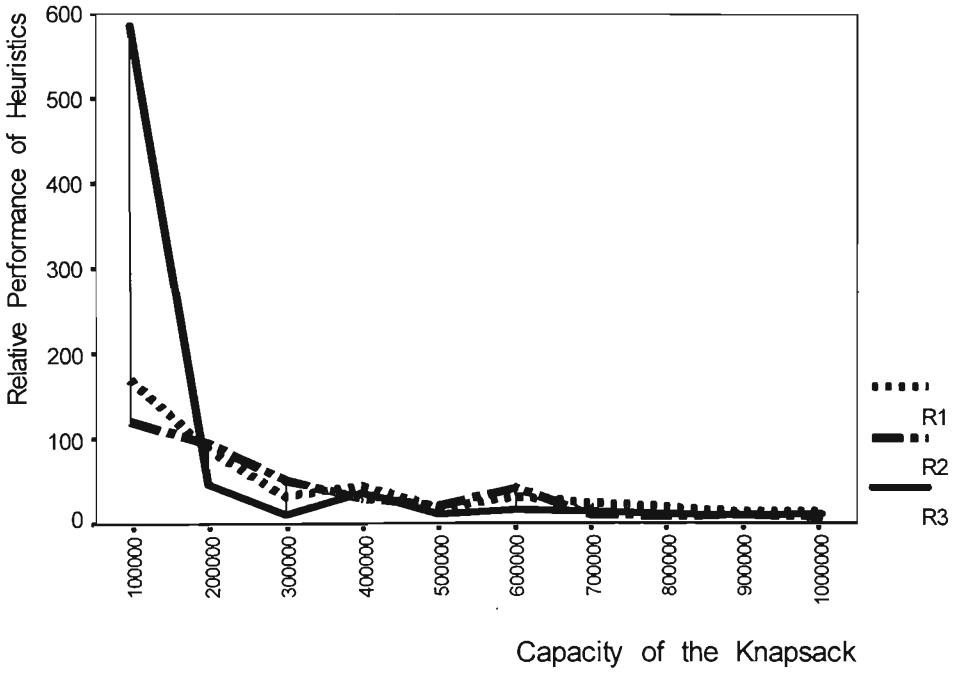

Figure 3.2 : Very Strongly Correlated Class (*Class IV*), n = 50 †

† The vertical scale has coded values. R1 = 170 refers to the case where the heuristic solution is only 0.000170% worse than the known optimal. The smaller the value of R1, R2 and R3, the closer the heuristic is to the optimal solution value.
$R1=10^6-(z(H_1)/z(opt))*10^6$, $R2=10^6-(z(H_2)/z(opt))*10^6$, $R3=10^6-(z(TV)/z(opt))*10^6$

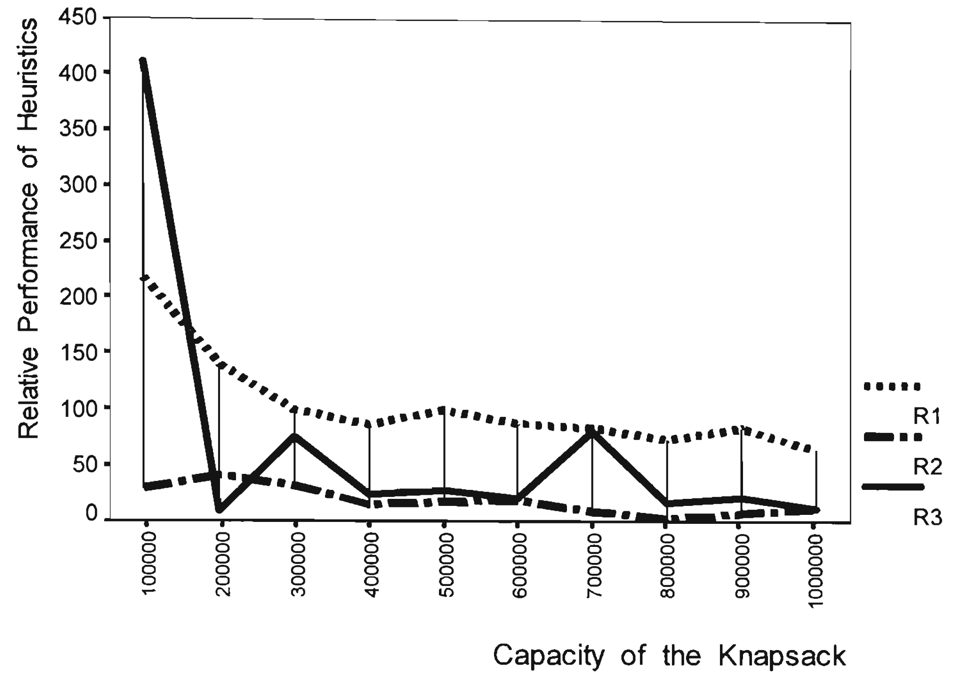

Figure 3.3 : Very Strongly Correlated Class (*Class IV*), n = 100

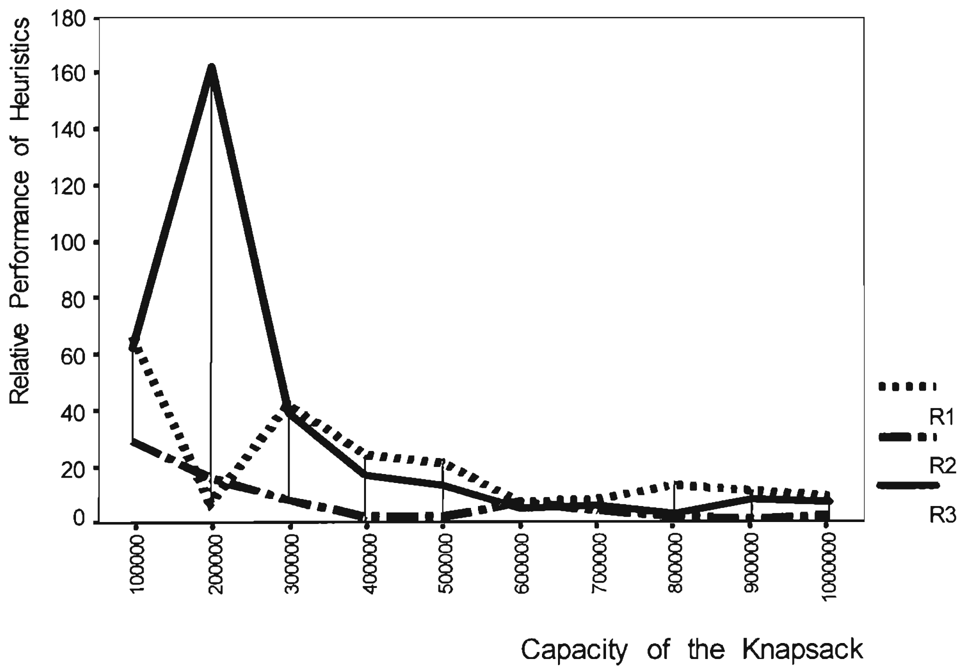

Figure 3.4 : Very Strongly Correlated Class (*Class IV*), n = 500

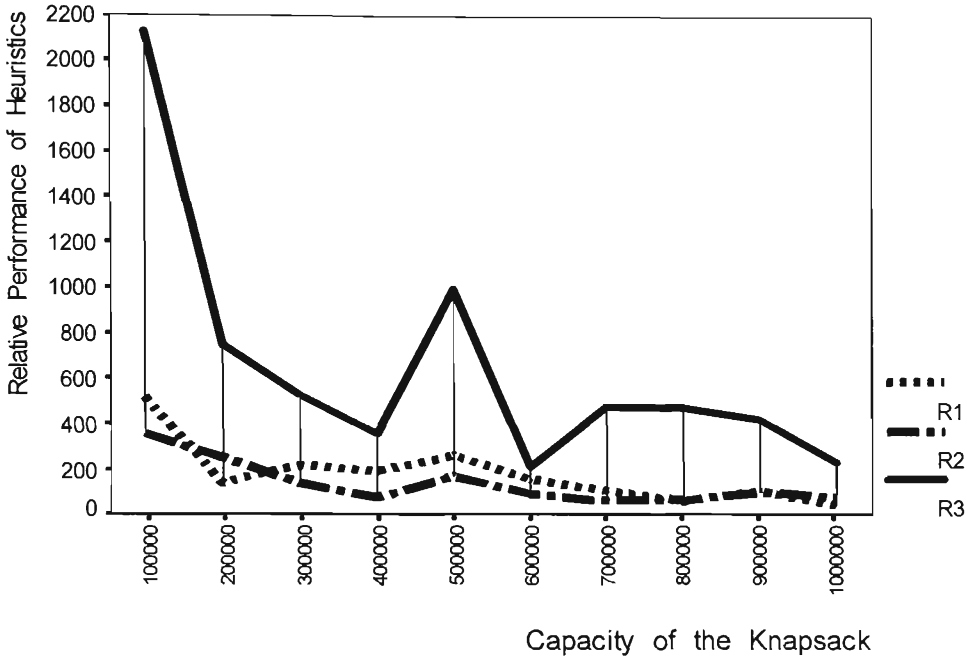

Figure 3.5 : Very Strongly Correlated Class (*Class IV*), n = 1000

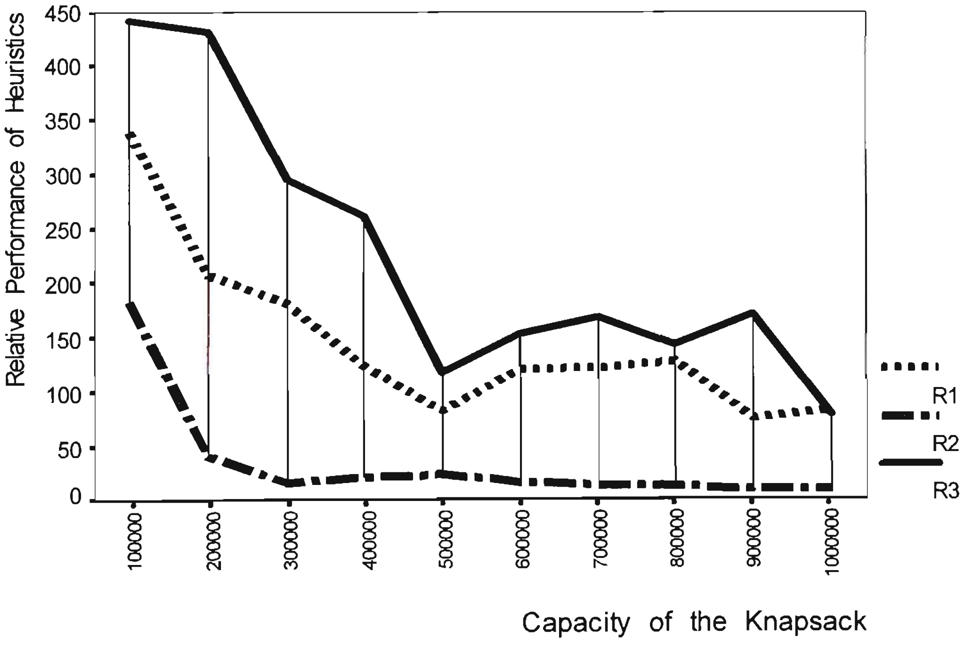

Figure 3.6 : Very Strongly Correlated Class (*Class IV*), n = 5000

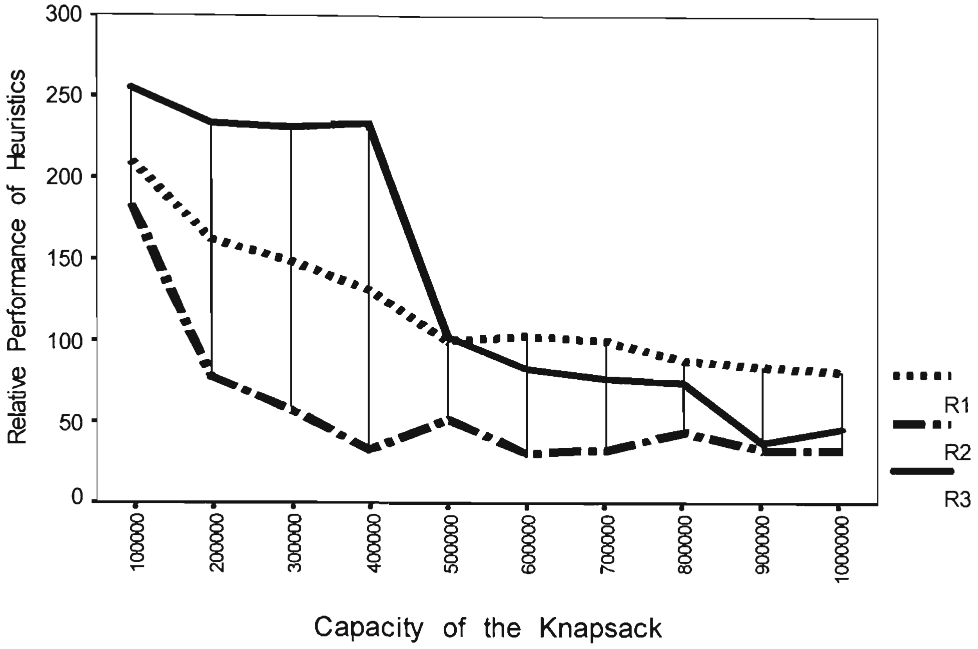

Figure 3.7 : Very Strongly Correlated Class (*Class IV*), n = 10000

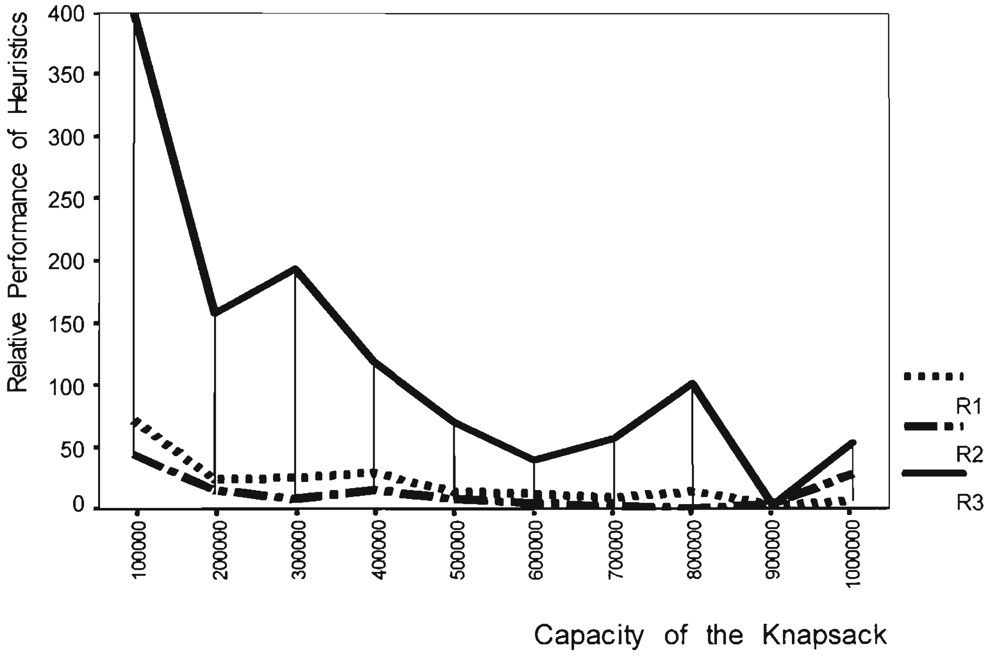

Figure 3.8 : Very Strongly Correlated Class (*Class IV*), n = 20000

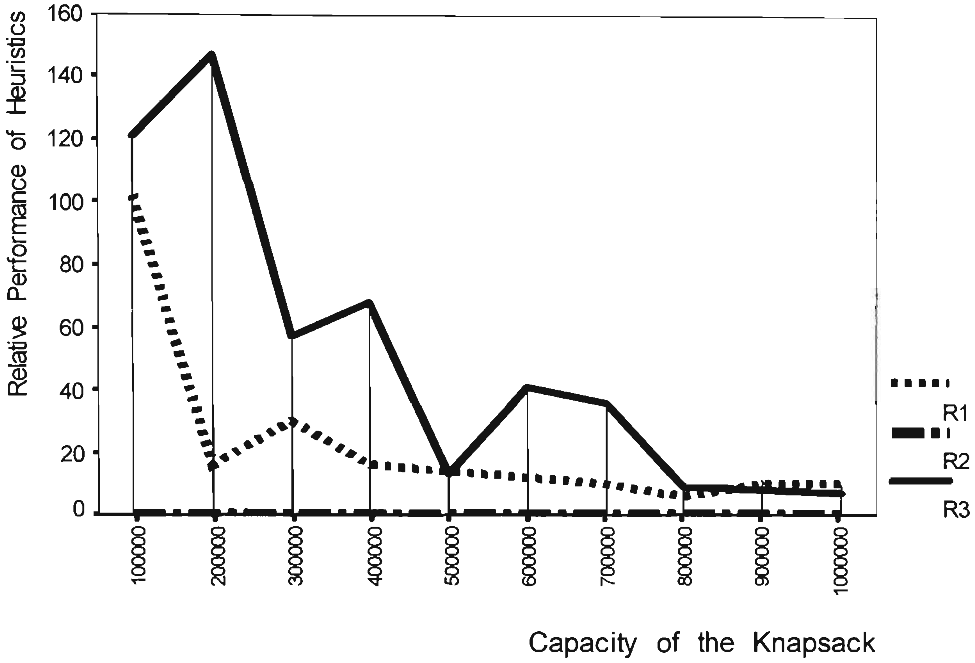

Figure 3.9 : Very Strongly Correlated Class (*Class IV*), n = 30000

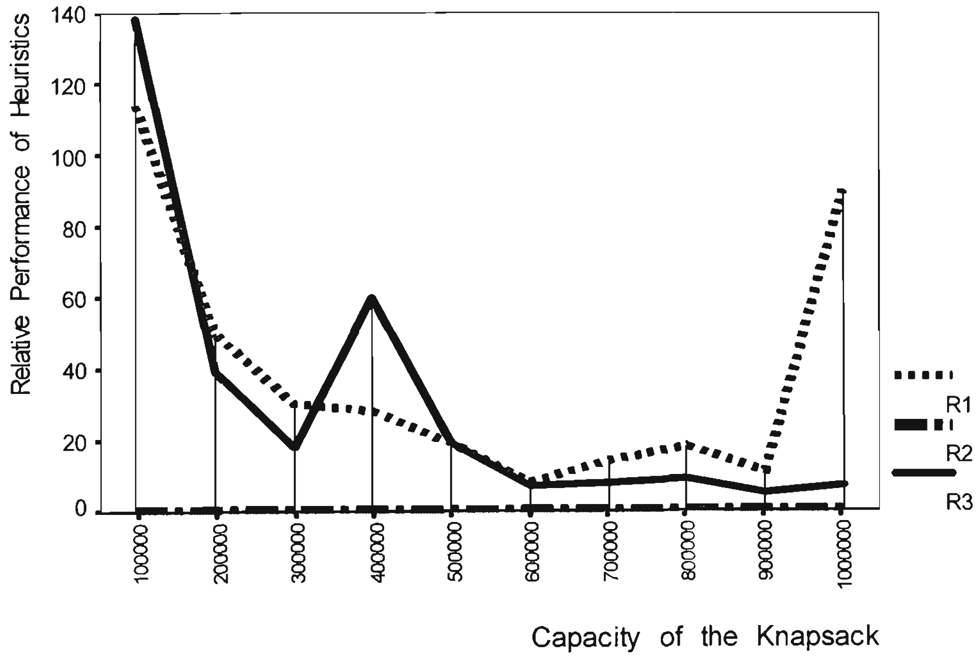

Figure 3.10 : Very Strongly Correlated Class (*Class IV*), n = 40000

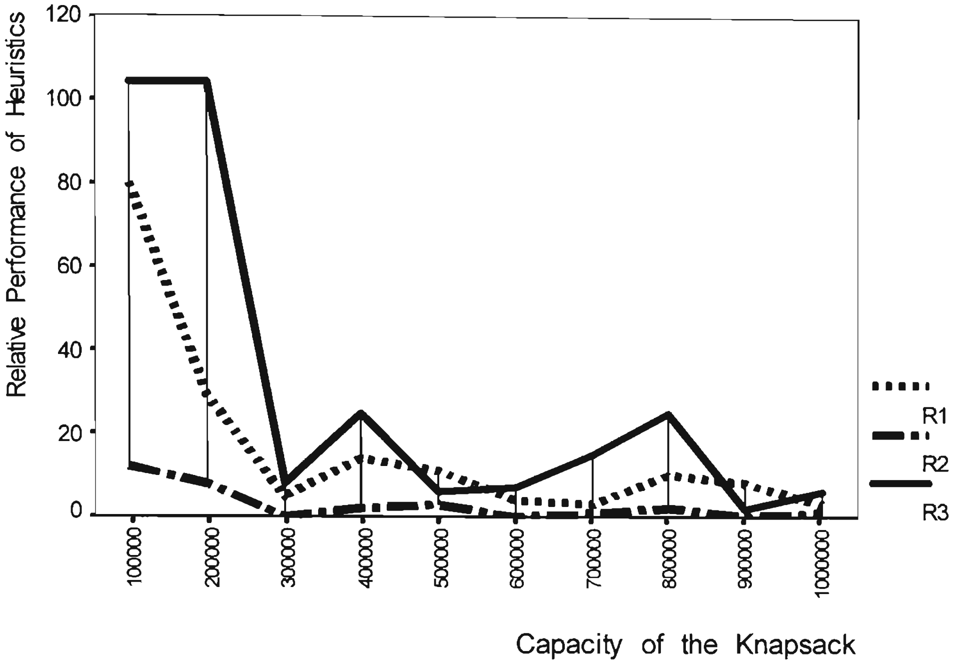

Figure 3.11 : Very Strongly Correlated Class (*Class IV*), n = 50000

For the *Class IV* problem instances, the extended greedy heuristic performs better than the other two heuristics for all problem sizes except for n = 50 where the total-value greedy heuristic performs better than the extended greedy heuristic for smaller capacities. Where total-value greedy fails, the extended greedy heuristic is good and where the extended greedy performs poorly, the total-value greedy heuristic performs better. As the capacity C of the knapsack increases, all the three heuristics come close to the optimal solution value.

The problems in *Class V* that are generated to give a large number of undominated items give the optimal solution value by all the three heuristics for large problem sizes, but for n = 100 and 500, the heuristics have varying performance. More specifically, for n = 100, the density-ordered greedy heuristic and the total-value greedy heuristic give near optimal solution value or sometimes the optimal solution value. Where these two heuristics fail, the extended greedy heuristic is better. For n = 500 and smaller knapsack capacities, density-ordered greedy and extended greedy performs better and the total-value heuristic performs poorly. As the capacity of the knapsack increases all the three heuristics perform well giving the optimal solution value.

The values of n and the problem classes for which there were no noticeable difference among the solution values given by the heuristics, have not been shown in the figures.

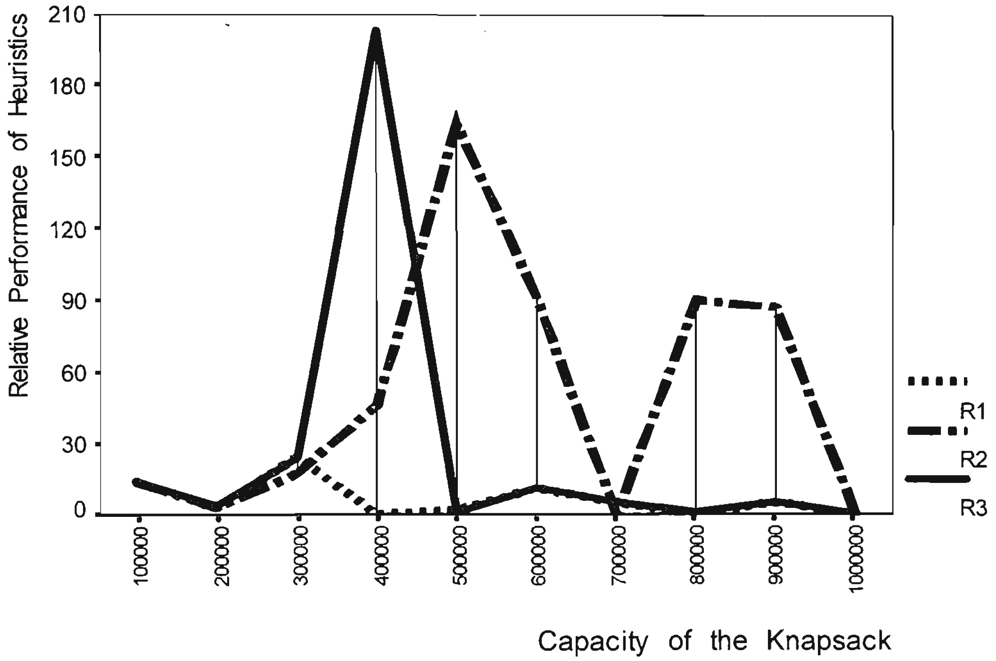

Figure 3.12 : Very Very Strongly Correlated Class (*Class V*), n = 100

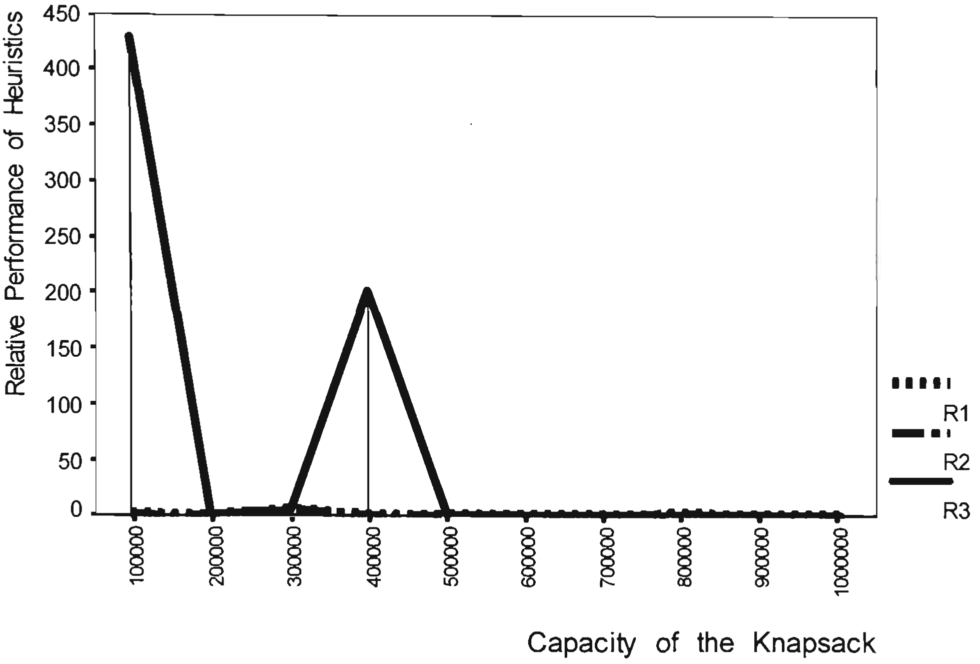

Figure 3.13 : Very Very Strongly Correlated Class (*Class V*), n = 500

CHAPTER 4

4. COMPLEMENTARY EFFECT OF HEURISTICS

As mentioned earlier, the total-value greedy heuristic has been shown to have a better worst-case bound result in comparison to the standard density-ordered greedy heuristic and can be used in a complementary mode to the density-ordered greedy heuristic, performing well where the density-ordered greedy heuristic performs poorly.

The joint performance of the density-ordered greedy and the total-value greedy heuristic has been discussed by Kohli and Krishnamurthi (1995). It has been shown that the combination gives a better performance result than the individual heuristics in the combination.

The following sections investigate the complementary effect of the density-ordered greedy, the extended greedy and the total-value greedy heuristics and also a new complementary heuristic that incorporates the structural properties of both the density-ordered greedy and the total-value greedy heuristic.

4.1 Comparison of Heuristics

An analysis was done to empirically investigate the complementary effect of the three efficient heuristic algorithms: H_1, H_2 and TV.

Table 4.1 presents the number of instances where each of the three heuristics (H_1, H_2 and TV) gives the best solution and also the number of instances where the combination of two heuristics, H_1 and H_2, H_1 and TV,

H_2 and TV and finally the combination of all the three is better than the individual heuristic.

This is applied to 2500 UKPs across the 5 problem classes for which reasonable lower bounds on the optimal solutions are known.

Table 4.1: Number of instances where the heuristics give optimal solution; 500 instances in each case

Problem Class	H_1	H_2	TV	$H_1 + H_2$	H_1 + TV	H_2 + TV	$H_1 + H_2$ + TV
Class I	335	335	383	335	383	383	383
Class II	400	413	399	421	400	421	421
Class III	374	494	310	494	374	494	494
Class IV	66	273	103	279	111	289	292
Class V	484	474	481	484	484	484	484
Total No. of Instances	1659	1989	1676	2013	1752	2071	2074
Percentage of problems solved optimally	66.36%	79.56%	67.04%	80.52%	70.08%	82.84%	82.96%

The complementary effect of H_1 and H_2 stands out for *Class II* and *Class IV* and the complementary effect of H_1 and TV is seen in *Class IV*. For instance, for Class II problem, 400 instances are solved optimally by H_1, 413 by H_2, but a combination of H_1 and H_2 solves 421 problem instances. H_2 and TV complements each other in *Class II, Class IV* and *Class V* and the complementary effect of the combination of all the three heuristics is substantial in *Class IV* though it is noticeable for the other four problem classes.

From the performance percentage figures of the three heuristics alone and their combination for the five problem classes, we see that the joint performance of one with H_2 outperforms the individual heuristics and the combination of all the three heuristics gives a much better performance result. This is mainly due to the involvement of H_2 in the combination. In some cases where H_2 performs rather poorly, each of H_1 and TV gives a better performance, thus complementing each other. However, as H_2 requires more computing time for large problem instances, particularly in *Class IV* and *Class V* problem instances, the combination of H_1 and TV can be recommended to solve a UKP within a reasonable computing time. It may be noted that Kohli and Krishnamurthi (1995) have theoretically shown that the worst-case performance of the combination of H_1 and TV is better than the worst-case performance of each heuristics in the combination. The empirical investigation undertaken in this study appears to support this. However, it must be stated that the strength of the complementary effect is not observed to be very high. Combining heuristics gave better results, but not always improved the results to optimality.

To further investigate the complementary effect of H_1 and TV, an algorithm that involves the characteristics of both these heuristics is developed. This heuristic can be called the *complementary total-value greedy heuristic* and is discussed in the following section.

4.2 Complementary Total-value Greedy Heuristic (CTVG)

This algorithm can be sketched as follows.

Step 1. Sort the items in the non-increasing order of the ratios.

Step 2. Find the total-value for every item in the sorted list.

Step 3. Choose the item with the largest total-value. Let this be the jth item of the sorted list.

Step 4. If $j = 1$, select item j.

If $j \neq 1$, find item k which has the second best total-value (it may be that the total-value of item j = total-value of item k).

Now, if $k = 1$, select item 1. If $k \neq 1$, then select item j.

Step 5. Update the knapsack capacity and the list of remaining items and repeat step 1 to step 4.

The FORTRAN implementation of this algorithm (CTVG) is given in Appendix B.

This complementary heuristic improves upon the individual heuristics in two respects.

The poor performance of the density-ordered greedy heuristic when the densest item leaves a significant capacity of the knapsack unused is somewhat compensated in each step by the total-value greedy heuristic that chooses items that fill more of the capacity and contribute more to the total solution value.

The total-value greedy heuristic's lack of ability to discriminate between the items that have the same total-value contribution with different densities is a demerit identified by Kohli and Krishnamurti (1995), the proposed heuristic overcomes this to some extent.

Example 1 is an instance where the new complementary total-value greedy heuristic is better than the other four greedy heuristics.

Example 1

C = 760

$w_5 = 210$, $w_4 = 90$, $w_3 = 80$, $w_2 = 60$, $w_1 = 65$

$p_5 = 690$, $p_4 = 260$, $p_3 = 230$, $p_2 = 170$, $p_1 = 175$

$\rho_5 = 690/210 > \rho_4 = 260/90 > \rho_3 = 230/80 > \rho_2 = 170/60 > \rho_1 = 175/65$

Iteration 1:

Ratio	w(j)	p(j)	Total-value		
3.286	210	690	2070	→	selected (capacity left = 130)
2.889	90	260	2080		
2.875	80	230	2070		
2.833	60	170	2040		
2.692	65	175	1925		

Iteration 2:

C = 130

Ratio	w(j)	p(j)	Total-value		
3.286	210	690	0		
2.889	90	260	260		
2.875	80	230	230		
2.833	60	170	340		
2.692	65	175	350	→	selected (capacity left = 0)

CTVG → $x_5 = 3$, $x_4 = 0$, $x_3 = 0$, $x_2 = 0$, $x_1 = 2$

z(CTVG) = 2420

TV → $x_5 = 0$, $x_4 = 8$, $x_3 = 0$, $x_2 = 0$, $x_1 = 0$

z(TV) = 2080

$H_1 \rightarrow x_5 = 3,\ x_4 = 1,\ x_3 = 0,\ x_2 = 0,\ x_1 = 0$

$z(H_1) = 2330$

$H_2 \rightarrow x_5 = 3,\ x_4 = 1,\ x_3 = 0,\ x_2 = 0,\ x_1 = 0$

$z(H_2) = 2330$

Optimal $\rightarrow x_5 = 3,\ x_4 = 0,\ x_3 = 0,\ x_2 = 0,\ x_1 = 2$

$z(opt) = 2420$

Example 2 below is an instance where the complementary total-value greedy heuristic performs as badly as the other heuristics.

Example 2

$C = 120$

$w_7 = 51,\ w_6 = 50,\ w_5 = 48,\ w_4 = 50,\ w_3 = 35,\ w_2 = 32,\ w_1 = 20$

$p_7 = 103,\ p_6 = 99,\ p_5 = 89,\ p_4 = 61,\ p_3 = 70,\ p_2 = 63,\ p_1 = 25$

The relative performance of the new complementary total-value greedy heuristic, the density-ordered greedy heuristic and the total-value greedy heuristic are discussed below. The weight-ordered greedy heuristic and the value-ordered greedy heuristic are not considered in the discussion of the performance of heuristics as both the heuristics perform poorly in comparison to the density-ordered greedy heuristic and the total-value greedy heuristic. The extended greedy heuristic is also ignored in the study of the complementary effect of heuristics as this heuristic requires more computational time for large problem instances.

The complementary total-value greedy heuristic is run on the existing 2500 data sets that were randomly generated as described in section 3.1, Chapter 3. All runs have been executed on the same 200MHz pentium Pro with option "-o" for the FORTRAN compiler.

Tables 4.2, 4.3, 4.4, 4.5 and 4.6 compare the complementary total-value greedy heuristic, the density-ordered greedy heuristic and the total-value greedy heuristic for UKP.

Table 4.2: Comparison of solutions for CTVG, H_1 and TV for *Class I* problems; 50 instances in each row (5 data sets and 10 different capacities)

Number of items, n	CTVG>H_1[*] CTVG>TV	H_1<CTVG =TV	CTVG<H_1 CTVG<TV	H_1>CTVG =TV	H_1=CTVG <TV	H_1=CTVG =TV
50	0	0	0	0	0	50 (40[†])
100	0	0	0	0	0	50 (50)
500	0	0	0	10	0	40 (34)
1000	0	0	0	0	0	50 (35)
5000	0	0	0	0	0	50 (20)
10000	0	0	0	0	0	50 (37)
20000	0	0	0	0	0	50 (28)
30000	0	0	0	0	0	50 (35)
40000	0	7	0	0	0	43 (43)
50000	0	0	0	0	0	50 (42)

Table 4.3: Comparison of solutions for CTVG, H_1 and TV for *Class II* problems; 50 instances in each row (5 data sets and 10 different capacities)

Number of items, n	CTVG>H_1 CTVG>TV	H_1<CTVG =TV	CTVG<H_1 CTVG<TV	H_1>CTVG =TV	H_1=CTVG <TV	H_1=CTVG =TV
50	0	0	0	0	0	50 (47)
100	0	0	0	0	0	50 (50)
500	0	0	0	0	0	50 (36)
1000	0	0	0	0	0	50 (44)
5000	0	0	0	0	0	50 (33)
10000	0	0	0	0	0	50 (43)
20000	0	0	0	0	0	50 (40)
30000	0	0	0	0	0	50 (36)
40000	0	0	0	0	0	50 (47)
50000	0	0	0	0	0	50 (50)

[*] CTVG > H_1 means the solution by CTVG is better than the solution by H_1. Other relations are similarly defined.

[†] The numbers in the bracket is the number of instances that gives the optimal solution value.

Table 4.4: Comparison of solutions for CTVG, H_1 and TV for *Class III* problems; 50 instances in each row (5 data sets and 10 different capacities)

Number of items, n	CTVG>H_1 CTVG>TV	H_1<CTVG =TV	CTVG<H_1 CTVG<TV	H_1>CTVG =TV	H_1=CTVG <TV	H_1=CTVG =TV
50	0	0	0	0	0	50 (34)
100	0	0	0	0	0	50 (16)
500	0	0	0	0	0	50 (23)
1000	0	0	0	0	0	50 (32)
5000	0	0	0	0	0	50 (30)
10000	0	0	0	0	0	50 (41)
20000	0	0	0	0	0	50 (27)
30000	0	0	0	0	0	50 (32)
40000	0	0	0	0	0	50 (36)
50000	0	0	0	0	0	50 (32)

Table 4.5: Comparison of solutions for CTVG, H_1 and TV for *Class IV* problems; 50 instances in each row (5 data sets and 10 different capacities)

Number of items, n	CTVG>H_1 CTVG>TV	H_1<CTVG =TV	CTVG<H_1 CTVG<TV	H_1>CTVG =TV	H_1=CTVG <TV	H_1=CTVG =TV
50	5	21 (1)	0	4	0	20 (13)
100	0	11	0	2	0	37 (13)
500	0	12	1	3	9	25
1000	6 (1)	5 (1)	0	23	0	16 (1)
5000	0	10 (2)	1	24	3 (1)	12
10000	3	12 (2)	1	16	2 (1)	16 (10)
20000	1	6	0	19	2	22 (11)
30000	2	13 (2)	2	26 (4)	4 (2)	3
40000	2	15 (8)	3	8 (1)	5 (3)	17 (4)
50000	0	13 (11)	0	15 (2)	1 (1)	21 (20)

Table 4.6: Comparison of solutions for CTVG, H_1 and TV for *Class V* problems; 50 instances in each row (5 data sets and 10 different capacities)

Number of	CTVG>H_1	H_1<CTVG	CTVG<H_1	H_1>CTVG	H_1=CTVG	H_1=CTVG
items, n	CTVG>TV	=TV	CTVG<TV	=TV	<TV	=TV
50	0	0	0	0	1 (1)	49 (35)
100	0	1	0	0	0	49 (40)
500	0	2	1	1	0	46 (40)
1000	0	0	0	1	0	49 (49)
5000	0	0	0	0	0	50 (50)
10000	0	0	0	0	0	50 (50)
20000	0	0	0	0	0	50 (50)
30000	0	0	0	0	0	50 (50)
40000	0	0	0	0	0	50 (50)
50000	0	0	0	0	0	50 (50)

To summarise the computational results of Tables 4.2 to 4.6 on the five problem classes, we look at the performance of the complementary total-value greedy heuristic in comparison to the density-ordered greedy heuristic and the total-value greedy heuristic.

For *Class I* (Table 4.2) there are 7 instances of the total 500 problem instances where CTVG performs as good as TV and in these instances H_1 performs poorly. In 10 problem instances H_1 outperforms CTVG which is as good as TV. The other 483 instances are cases where CTVG performs equally good as H_1 and TV of which for 364 instances an optimal solution value is obtained.

In *Class II* (Table 4.3) and *Class III* (Table 4.4), all the three heuristics perform equally well of which 426 instances in *Class II* and 303 instances in *Class III* gives the optimal solution value.

In *Class IV* (Table 4.5), CTVG outperforms H_1 and TV in 19 instances (of which the optimal solution is reached by 1 problem instance) and performs poorly in 8 problem instances. There are 118 problem instances (optimal solution value is found by 27 instances) which are solved by CTVG giving solutions equal to TV and better than H_1 and 140 instances (7 problem instances giving the optimal solution value) are solved by H_1 whose solution value is better than that of CTVG and TV. There are also 26 instances where CTVG performs as well as H_1 but poorly in comparison to TV. In 189 problem instances, CTVG performs as good as the other two heuristics with 72 instances giving the optimal solution value. The usefulness of CTVG is best illustrated in *Class IV* problems.

In *Class V* (Table 4.6) problems, CTVG outperforms H_1 in 3 instances and H_1 outperforms CTVG in 2 instances. There is just 1 instance where CTVG is poor in comparison to both H_1 and TV. 493 problem instances are solved by all the three heuristics equally well of which 464 instances give the optimal solution value.

Table 4.7 presents the number of instances where each of H_1, H_2, TV and CTVG give the best solution and also the number of instances where the combination of three H_1, H_2 and TV and the combination of all four H_1, H_2, TV and CTVG is better than the individual heuristic.

Table 4.7: Number of instances where the heuristics give optimal solution; 500 instances in each case

Problem Class	H_1	H_2	TV	CTVG	$H_1 + H_2 +$ TV	$H_1 + H_2 +$ TV + CTVG
Class I	335	335	383	364	383	383
Class II	400	413	399	426	421	426
Class III	374	494	310	303	494	494
Class IV	66	273	103	100	292	292
Class V	484	474	481	464	484	484
Total No. of Instances	1659	1989	1676	1657	2074	2079
Percentage of problems solved optimally	66.36%	79.56%	67.04%	66.28%	82.96%	83.16%

We see that the performance percentage of the individual heuristic is improved when the combination of all the four heuristics are considered.

To conclude we can see that no heuristic show any clear superiority in performance but they complement each other. Thus, while none of the heuristics takes a lot of computing time, they perform as well as the exact solution algorithm in most problem classes and where they fail to perform individually, the performance is bettered by considering the combination of the heuristics. Therefore, any of the heuristics by itself or in combination with another can be used to solve any large Unbounded Knapsack Problem or at least provide a lower bound for it.

CHAPTER 5

5. SUMMARY AND CONCLUSION

5.1 Summary

An investigation has been done on the performance of the five greedy heuristics that have been suggested in the literature for the Unbounded Knapsack Problem. Though Martello-Toth algorithm is good in finding the optimal solution to an Unbounded Knapsack Problem, the heuristic algorithms are much faster giving near optimal solution values. This is largely because of the greedy structure of the algorithm. The phenomenon of dominance plays a very important role; the problem size is reduced by dominance criterion resulting in efficient solution. The performance of the heuristics is studied by varying parameters such as the number of items, knapsack capacity and the density ratios.

As expected for any combinatorial optimisation problem, the larger the number of items, the more difficult an UKP is to solve. However, the phenomenon of dominance necessitates redefining this statement as: *the larger the number of undominated items, the more difficult it is to solve*. This pattern was consistently seen in all the problem instances.

Regarding the capacity of the knapsack, it was observed that the larger the capacity, the smaller the difference between the optimal solution and a heuristic solution. This is an expected result as with larger capacities, the difference between the z-values for one item or another is not appreciable. The knapsack capacity indirectly takes into account the parameter k, the maximum number of the largest item that can be assigned to a knapsack. As Kohli and Krishnamurthi (1992) showed, with larger k, the difference between the worst-case bounds of the density-ordered greedy and the total-value greedy algorithms diminishes.

As regards the density ratios (p/w ratios), the ratios themselves do not influence the solution quality directly. However, if the density ratios of the items are all in a narrow range, it is more likely that there would be a large number of undominated items and thus the solution would be relatively difficult. This was the case in our study. The ratio range was particularly relevant in *Class IV* type of problems, where it was observed that for smaller ratio ranges, H_1 and H_2 performs well and come close to the optimal solution value as the capacity increases and for larger ratio ranges, TV gives the optimal solution value where as H_1 and H_2 performs poorly.

Computational analysis of the five heuristics shows that H_2 outperforms the other four greedy heuristics in most problem instances and where H_2 fails, in some cases it is either H_1 or TV performing better. Table 4.1 shows that with the combination of H_1, H_2 and TV, a better performance result can be obtained. Since the computational time requirement of H_2 is the highest for large problem instances, particularly in the *Class IV* and *Class V* problems, the combination of H_1 and TV can be looked at. Kohli and Krishnamurti (1995) have theoretically proved that the worst-case performance of the combination of H_1 and TV is better than the worst-case performance of the single best heuristic in the combination. This is supported by the empirical findings of this study.

The suggested combination of H_1 and TV was further studied by developing an algorithm that combines the characteristics of both the individual heuristics. This complementary heuristic was called the complementary total-value greedy heuristic (CTVG).

CTVG was executed on the same data sets and it was found that this heuristic performs as well as the other two heuristics in most problem instances and outperforms both H_1 and TV only in a few instances. It should also be mentioned that CTVG performs poorly for a few instances. Thus, it can be said that though CTVG does not show any clear superiority in the relative performance for the five problem classes, the combination of CTVG with H_1, H_2 and TV improves the performance result.

5.2 Conclusion

In this thesis, we have analysed the performance of the five greedy heuristics that have been suggested in the literature for the unbounded knapsack problem. It has been found that all the heuristics perform well in comparison to the optimal solution algorithm. The running time for the density-ordered greedy heuristic, the total-value greedy heuristic, the weight-ordered greedy heuristic and the value-ordered greedy heuristic is negligible in comparison to the exact solution which takes a few hundred seconds for large problems in *Class IV* and takes a few hours for problem instances of *Class V*, particularly when the problem size n is around 500. The extended greedy heuristic takes a few seconds for large problems, but often gives better solutions than other heuristics.

By generating problem instances that can give many undominated items, the difficulty of solving the problems was increased. For these difficult problem instances, the heuristic solutions were not far from the optimal solutions.

It can be concluded that H_1, H_2, TV and CTVG perform well in comparison to the exact solution algorithm and they exhibit some complementary effect. As has been suggested by White (1992), combining heuristics is a good method for solving hard combinatorial problems. Also, using a number of cheap heuristics to calculate bounds for a hard combinatorial problem is a sound practice (Davies, 1974). The investigation in this study confirms this for the Unbounded Knapsack Problem.

5.3 Further Scope for Research

1. Meta-heuristics, like genetic algorithm, simulated annealing or tabu search are the most recent development in approximate search methods for solving complex optimisation problems. These are designed to attack hard combinatorial optimisation problems where classical heuristics have failed to be effective and efficient. Meta-heuristic search methods are used in many applications including the 0-1 Knapsack Problems but so far very little, if any, has been done for the Unbounded Knapsack Problem. Although the suggested heuristics for the Unbounded Knapsack Problem are fast and give close to optimal solutions, meta-heuristics can definitely be looked at because of its iterative generation process approach.

2. A combination of the density-ordered greedy heuristic and the total-value greedy heuristic (*complementary total-value greedy heuristic, CTVG*) was shown to improve the performance of the heuristic in comparison to the individual heuristic in the combination. More algorithms of this type can be developed and evaluated.

3. Characterisation of the problem instances for the complementary total-value greedy heuristic and other heuristics of this kind with respect to the difficulty of the solution is worth considering. Also the relation between the heaviest item (largest w) that can fit into the knapsack, and the difficulty of solving UKP can be investigated.

APPENDIX

Appendix A : FORTRAN Codes for Data Generation

The following are the FORTRAN codes for generating data for the Uncorrelated (*Class I*), Weakly Correlated (*Class II*), Strongly Correlated (*Class III*), Very Strongly Correlated (*Class IV*) and Very Very Strongly Correlated (*Class V*) Class of problems.

Uncorrelated (*Class I*)

```
c   program uncorr.for
c   program to create pairwise uncorrelated data randomly
c   w(i) is uniformly random in [1, 9999]
c   p(i) is uniformly random in [1, 9999]
       real p(50000), w(50000), ratio
       character * 15 dfname1
       write (*, *) ' write the name of the datafile to be generated: '
       read (*, 55) dfname1
55   format (a15)
       open (unit = 9, file = dfname1)
       write (*, *) ' write the number of datapairs:  '
       read  (*, *) n
       randm = rrand()
       write (*, *)  ' write the range of data: p-values (1, 9999) '
       read  (*, *) iminp, imaxp
       write (*, *)  ' write the range of data: w-values (1, 9999) '
       read  (*, *) iminw, imaxw
       do 51 i = 1, n
41     p(i) =  int(rnd() * imaxp)
         if (p(i) .lt. iminp) goto 41
42     w(i) =  int(rnd() * imaxw)
         if (w(i) .lt. iminw) goto 42
         ratio = float (p(i) / w(i))
         write (9,101) ratio, w(i), p(i)
51     continue
101    format (f7.3,  i12,  2(2x,i10),  4x)
       close (unit = 9, file = dfname1)
       stop
       end
```

Weakly Correlated *(Class II)*

```
c    program wkcorr.for
c    program to create weakly correlated data randomly
c    w(i) is uniformly random in [10, 9999]
c    p(i) is uniformly random in [w(i) - 100, w(i) + 100]
         real p(50000), w(50000), ratio
         character * 15 dfname1
         write (*, *) ' write the name of the datafile to be generated:  '
         read (*, 55) dfname1
55    format (a15)
         open (unit = 9, file = dfname1)
         write (*, *) ' write no. of datapairs:  '
         read  (*, *) n
         randm = rrand()
         write (*, *) ' write the range of data: w-values (10, 9999)'
         read  (*, *) iminw, imaxw
         do 51 i = 1, n
41      w(i) =  int(rnd() * imaxw)
           if (w(i) .lt. iminw) goto 41
           iminp = w(i) - 100
           imaxp = w(i) + 100
42      p(i) = int(rnd() * imaxp)
           if (p(i) .lt. iminp) goto 42
           ratio = float (p(i) / w(i))
           if (ratio .eq. 0) goto 41
           write (9,101) ratio, w(i), p(i)
51    continue
101   format (f7.3,  i12,  2(2x, i10),  4x)
         close (unit = 9, file = dfname1)
         stop
         end
```

Strongly Correlated *(Class III)*

```
c    program sgcorr.for
c    program to create strongly correlated data randomly
c    w(i) is uniformly random in [10, 9999]
c    p(i) = w(i) + 100
         real p(50000), w(50000), ratio
         character * 15 dfname1
         write (*, *) ' write the name of the datafile to be generated:  '
         read (*, 55) dfname1
55    format (a15)
         open (unit = 9, file = dfname1)
```

```
        write (*, *) ' write no. of datapairs:  '
        read  (*, *) n
        randm = rrand()
        write (*, *) 'write range of data: w-values (10, 9999)'
        read  (*, *) iminw, imaxw
        do 51 i = 1, n
41      w(i) =  int(rnd() * imaxw)
          iw = w(i)
          if (w(i) .lt. iminw .or. w(i) .gt. imaxw) goto 41
          p(i) =  iw + 100
          ratio = float (p(i) / w(i))
          write (9,101) ratio, w(i), p(i)
51    continue
101   format (f7.3,  i12,  2(2x, i10), 4x)
      close (unit = 9, file = dfname1)
      stop
      end
```

Very Strongly Correlated *(Class IV)*

```
c     program vscorrb.for
c     program to create very strongly correlated data randomly
c     w(i) is uniformly random in [1, 9999]
c     p(i) is uniformly random in [1, 9999]
c     2 <= p(i)/w(i) <= 2.5
        real p(50000), w(50000), ratio
        real min_rat, max_rat
        character * 15 dfname1
        write (*, *) ' write the name of the datafile to be generated :  '
        read (*, 55) dfname1
55    format (a15)
        open (unit = 9, file = dfname1)
        write (*, *) ' write no. of datapairs:  '
        read  (*, *) n
        randm = rrand()
        write (*, *) ' write range of data: p-values (1, 9999)'
        read  (*, *) iminp,imaxp
        write (*, *) ' write the range of the ratios (e.g., 2.0, 2.5)  '
        read (*, *) min_rat, max_rat
        do 51 i = 1, n
42      p(i) =  int(rnd() * 10000)
          w(i) =  int(rnd() * 10000)
          if (p(i) .lt. iminp .or. p(i) .gt. imaxp) goto 42
          if (w(i) .lt. 1) goto 42
          ratio = float (p(i) / w(i))
          if (ratio .le. max_rat .and. ratio .ge. min_rat) then
                write (9,101) ratio, w(i), p(i)
          else
                go to 42
          end if
51     continue
101   format (f7.3, i12, 2(2x,i10), 4x)
        close (unit = 9, file = dfname1)
        stop
        end
```

Very Very Strongly Correlated (*Class V*)

```
c    program vvscorr.for
c    program to create very very strongly correlated data randomly
c    w(i) is uniformly random in [1, 99999]
c    p(i) = w(i) * ((w(i) - wmin + 1) / (wmax - wmin + 1)) * 100
        real p(50000), w(50000), ratio
        character * 15 dfname1
        write (*, *) ' write the name of the datafile to be generated  :  '
        read (*, 55) dfname1
55    format (a15)
        open (unit = 9, file = dfname1)
        write (*, *) 'write no. of datapairs:  '
        read  (*, *) n
        randm = rrand()
        write (*, *) 'write range of data: w-values (1, 99999)'
        read  (*, *) iminw,imaxw
        do 51 i = 1, n
42      w(i) =  int(rnd() * imaxw)
          if (w(i) .lt. iminw .or. w(i) .gt. imaxw) goto 42
          if (w(i) .lt. 1) goto 42
          p(i) = w(i) * ((w(i)-iminw+1) / (imaxw-iminw+1)) * 100
          if (p(i) .lt. 1) goto 42
          ratio = float (p(i) / w(i))
          write (9,101) ratio, w(i), p(i)
51     continue
101   format (f7.3, i12, 2(2x,i10), 4x)
        close (unit = 9, file = dfname1)
        stop
        end
```

Appendix B : FORTRAN Implementations of the Heuristic Algorithms

The following are the FORTRAN implementations of the five heuristic algorithms as described in Chapter 3, Section 3.4.

The Fortran code for dominance check is included.

Sorting of the items is done by bubble sort.

DGREEDY (Density Ordered Greedy Heuristic)

```
c   icheck = a check for elimination of dominated items.
c   if icheck = 1 or more, the input data will go through a stage
c   of elimination of dominated items.
      integer  w(50000), p(50000)
      integer low_c, high_c, incre
      real ratio(50000)
      character * 15 dfname1
      character * 15 dfname2
      character * 30 string
      print *, 'input the number of items, n: '
      read *, n
      if (n .gt. 50000) stop 01
      write (*, *) 'write the name of the datafile :'
      read (*, 2) dfname1
      open (unit = 2, file = dfname1)
2     format (a15)
      do 3 i = 1, n
      read (2, '(f7.3, 2i12)')  ratio(i), w(i), p(i)
3     continue
      rewind unit = 2
      close (unit = 2)
      write (*, *) 'write the name of the output file :'
      read (*, 4) dfname2
4     format (a15)
      write (*, *) 'do you like to check for dominance? 1(yes), 0(no):'
      read (*, *) icheck
      call fdate (string)
      write (*, *) ' starting time :  ', string
      if (icheck .ge. 1) call domcheck (n, ratio, w, p)
      call fdate (string)
      write (*, *) ' finished time : ', string
      write (*, *) 'would you like to sort by density (decreasing) the
     +datafile? 1(yes), 0(no) :'
      read (*, *) num
      call fdate (string)
      write (*, *) ' starting time : ', string
      if (num .eq. 1) call dsort (n, ratio, w, p)
      call fdate (string)
      write (*, *) ' finished time : ', string
      print *, 'input the capacity range low-c, high-c of the knapsack:'
      read *, low_c, high_c
      print *, 'input the increment for the capacity: '
      read *, incre
      call fdate (string)
      write (*, *) ' starting time : ', string
      do 11 ic = low_c, high_c, incre
11      call greedy (n, w, p, ic, ratio, dfname2)
      close (unit = 3)
      end

      subroutine greedy (n, w, p, ic, ratio, dfname2)
      integer w(50000), p(50000), ow(50000), op(50000), c_left
      integer x(50000)
      real ratio(50000), oratio(50000)
      character * 30 string
      character * 15 dfname2
      integer ctr
      do 10 j = 1, n
      ow(j) = w(j)
      op(j) = p(j)
      oratio(j) = ratio(j)
10     continue
      open (unit = 3, file = dfname2)
        c_left = ic
      write (3, 20) n , ic
        ctr = 0
      do 12 j = 1, n
        ctr = ctr + 1
        x(ctr) = int(c_left / ow(j))
        c_left = c_left - x(ctr) * ow(j)
12      continue
        t = 0
      do 13 j = 1, n
        t = t + (x(j) * op(j))
13      continue
```

```
      write (*, *) t
      call fdate (string)
      write (*, *) ' finished time : ', string
      write (3, 30)
20    format (  '  density-ordered greedy heuristic ;',
     +'  n (undominated) =' i6, ';   ', 'c = 'i8)
30    format ('      item     number  weight  profit  ratio')
      do 14 j = 1, n
      if (x(j) .lt. 1) goto 14
      write (3, '(2x, 4i10, 2x, f7.3)') j, x(j), ow(j), op(j), oratio(j)
14    continue
      write (3, 40)
40    format ( '  z(h1) =  ')
      write (3, *) t
      write (3, 50)
50    format ( 53h          ---------------------------------        )
      return
      end
```

WGREEDY (Weight Ordered Greedy Heuristic)

```
c     icheck = a check for elimination of dominated items.
c     if icheck = 1 or more, the input data will go through a stage
c     of elimination of dominated items.
      integer  w(50000), p(50000)
      integer low_c, high_c, incre
      real ratio(50000)
      character * 15 dfname1
      character * 15 dfname2
      character * 30 string
      print *, 'input the number of items, n: '
      read *, n
      if (n .gt. 50000) stop 01
      write (*, *) 'write the name of the datafile :'
      read (*, 2) dfname1
      open (unit = 2, file = dfname1)
2     format (a15)
      do 3 i = 1, n
      read (2, '(f7.3, 2i12)')  ratio(i), w(i), p(i)
3     continue
      rewind unit = 2
      close (unit = 2)
      write (*, *) 'write the name of the output file :'
      read (*, 4) dfname2
```

```
4     format (a15)
      write (*, *) 'do you like to check for dominance? 1(yes), 0(no):'
      read (*, *) icheck
      call fdate (string)
      write (*, *) ' starting time :  ', string
      if (icheck .ge. 1) call domcheck (n, ratio, w, p)
      call fdate (string)
      write (*, *) ' finished time : ', string
      write (*, *) 'would you like to weight-sort (acsending) the
     +datafile? 1(yes), 0(no):'
      read (*, *) num
      call fdate (string)
      write (*, *) ' starting time : ', string
      if (num .ge. 1) call wtsort (n, ratio, w, p)
      call fdate (string)
      write (*, *) ' finished time : ', string
      print *, 'input the capacity range low-c, high-c of the knapsack:'
      read *, low_c, high_c
      print *, 'input the increment for the capacity: '
      read *, incre
      call fdate (string)
      write (*, *) ' starting time : ', string
      do 11 ic = low_c, high_c, incre
11      call wtgreedy (n, w, p, ic, ratio, dfname2)
      close (unit = 3)
      end

      subroutine wtgreedy (n, w, p, ic, ratio, dfname2)
      integer w(50000), p(50000), ow(50000), op(50000), c_left
      integer x(50000)
      real ratio(50000), oratio(50000)
      character * 30 string
      character * 15 dfname2
      integer ctr
      do 10 j = 1, n
      ow(j) = w(j)
      op(j) = p(j)
      oratio(j) = ratio(j)
10      continue
      open (unit = 3, file = dfname2)
        c_left = ic
      write (3, 20) n , ic
        ctr = 0
      do 12 j = 1, n
        ctr = ctr + 1
```

```
         x(ctr) = int(c_left / ow(j))
         c_left = c_left - x(ctr) * ow(j)
 12      continue
         t = 0
      do 13 j = 1, n
         t = t + (x(j) * op(j))
 13      continue
      write (*, *) t
      call fdate (string)
      write (*, *) ' finished time : ', string
      write (3, 30)
 20    format (  '  weight-ordered greedy heuristic ;',
     +'  n (undominated) =' i6, ';    ', 'c = 'i8)
 30    format ('      item     number   weight   profit   ratio')
      do 14 j = 1, n
      if (x(j) .lt. 1) goto 14
      write (3, '(2x, 4i10, 2x, f7.3)') j, x(j), ow(j), op(j), oratio(j)
 14    continue
      write (3, 40)
 40    format ( '  z(h3) =  ')
      write (3, *) t
      write (3, 50)
 50    format ( 53h          ----------------------------------        )
      return
      end
```

VGREEDY (Value Ordered Greedy Heuristic)

```
c    icheck = a check for elimination of dominated items.
c    if icheck = 1 or more, the input data will go through a stage
c    of elimination of dominated items.
      integer  w(50000), p(50000)
      integer low_c, high_c, incre
      real ratio(50000)
      character * 15 dfname1
      character * 15 dfname2
      character * 30 string
      print *, 'input the number of items, n: '
      read *, n
      if (n .gt. 50000) stop 01
      write (*, *) 'write the name of the datafile :'
      read (*, 2) dfname1
      open (unit = 2, file = dfname1)
 2    format (a15)
```

```
      do 3 i = 1, n
      read (2, '(f7.3, 2i12)')  ratio(i), w(i), p(i)
 3     continue
      rewind unit = 2
      close (unit = 2)
      write (*, *) 'write the name of the output file :'
      read (*, 4) dfname2
 4     format (a15)
      write (*, *) 'do you like to check for dominance? 1(yes), 0(no):'
      read (*, *) icheck
      call fdate (string)
      write (*, *) ' starting time :  ', string
      if (icheck .ge. 1) call domcheck (n, ratio, w, p)
      call fdate (string)
      write (*, *) ' finished time : ', string
      write (*, *) 'would you like to sort the datafile by profit
     +(descending)? 1(yes), 0(no):'
      read (*, *) num
      call fdate (string)
      write (*, *) ' starting time : ', string
      if (num .ge. 1) call vlsort (n, ratio, w, p)
      call fdate (string)
      write (*, *) ' finished time : ', string
      print *, 'input the capacity range low-c, high-c of the knapsack:'
      read *, low_c, high_c
      print *, 'input the increment for the capacity: '
      read *, incre
      call fdate (string)
      write (*, *) ' starting time : ', string
      do 11 ic = low_c, high_c, incre
 11    call vlgreedy (n, w, p, ic, ratio, dfname2)
      close (unit = 3)
      end

      subroutine vlgreedy (n, w, p, ic, ratio, dfname2)
      integer w(50000), p(50000), ow(50000), op(50000), c_left
      integer x(50000)
      real ratio(50000), oratio(50000)
      character * 30 string
      character * 15 dfname2
      integer ctr
      do 10 j = 1, n
      ow(j) = w(j)
      op(j) = p(j)
      oratio(j) = ratio(j)
```

```
10   continue
     open (unit = 3, file = dfname2)
      c_left = ic
     write (3, 20) n , ic
      ctr = 0
     do 12 j = 1, n
      ctr = ctr + 1
      x(ctr) = int(c_left / ow(j))
      c_left = c_left - x(ctr) * ow(j)
12   continue
      t = 0
     do 13 j = 1, n
      t = t + (x(j) * op(j))
13   continue
     write (*, *) t
     call fdate (string)
     write (*, *) ' finished time : ', string
     write (3, 30)
20   format (  '  value-ordered greedy heuristic ;',
    +'  n (undominated) =' i6, ';    ', 'c = 'i8)
30   format ('       item     number   weight   profit   ratio')
     do 14 j = 1, n
     if (x(j) .lt. 1) goto 14
     write (3, '(2x, 4i10, 2x, f7.3)') j, x(j), ow(j), op(j), oratio(j)
14   continue
     write (3, 40)
40   format ( '  z(h4) =  ')
     write (3, *) t
     write (3, 50)
50   format ( 53h          ---------------------------------        )
     return
     end
```

EXTGREED (Extended Greedy Heuristic)

```
c    w = weight of an item.
c    p = profit of an item.
c    low_c = lowest capacity.
c    high_c = highest capacity.
c    incre = increment in the capacity.
c    ratio = profit/weight of an item.
c    dfname1 = name of the datafile.
c    n = number of items.
c    num = a check for sorting the undominated items.
c    if num = 1 or more, the undominated items will be sorted in
```

```
c     decreasing order of the ratios.
c     ic = incremented capacity.
c     ismall = smallest item weight.
c     icheck = a check for elimination of dominated items.
c     if icheck = 1 or more, the input data will go through a stage
c     of elimination of dominated items.
      integer  w(50000), p(50000)
      integer low_c, high_c, incre
      real ratio(50000)
      character * 15 dfname1
      character * 15 dfname2
      character * 30 string
      print *, 'input the number of items, n: '
      read *, n
      if (n .gt. 50000) stop 01
      write (*, *) 'write the name of the datafile :'
      read (*, 2) dfname1
      open (unit = 4, file = dfname1)
 2    format (a15)
      do 3 i = 1, n
      rewind unit =4
      read (4, '(f7.3, 2i12)' ) ratio(i), w(i), p(i)
 3    continue
      close (unit = 4)
      write (*, *) 'write the name of the output file :'
      read (*, 9) dfname2
 9    format (a15)
      write (*, *) 'do you like to check for dominance? 1(yes), 0(no):'
      read (*, *) icheck
      call fdate (string)
      write (*, *), '  starting time :  ' , string
      if (icheck .ge. 1) call domcheck (n, ratio, w, p, ismall)
      call fdate (string)
      write (*, *), '  finished time :  ' , string
      write (*, *) 'would you like to sort the datafile? 1(yes), 0(no):'
      read (*, *) num
      if (num .ge. 1) call sort (n, ratio, w, p)
      print *, 'input the capacity range low-c, high-c of the knapsack:'
      read *, low_c, high_c
      print *, 'input the increment for the capacity: '
      read *, incre
      call fdate (string)
      write (*, *), ' starting time : ' , string
      do 11 ic = low_c, high_c, incre
 11   call extgreed (n, w, p, ic, ismall, ratio, dfname2)
```

```
      end

      subroutine extgreed (n, w, p, ic, ismall, ratio, dfname2)
c     n = number of items in the knapsack.
c     w, ow = weight of an item.
c     p, op = profit of an item.
c     x = number of units of an item selected.
c     c_left = capacity left.
c     ic = incremented capacity.
c     ismall = smallest item weight.
      integer w(50000), p(50000), ow(50000), op(50000), x(50000)
      integer c_left
      real ratio(50000), oratio(50000)
      character * 15 dfname2
      character * 30 string
      do 10 m = 1, n
      ow(m) = w(m)
      op(m) = p(m)
      oratio(m) = ratio(m)
10      continue
      m = n
      if (mod (n, 2) .ne. 0) then
            m = m + 1
            ow(m) = 999
            op(m) = 1
      end if
      open (unit = 7, file = dfname2)
        c_left = ic
      write (7, 20) n, c_left
        profit = 0
        store = 0
      do 12 mm = 1, m - 1, 2
        mm1 = mm + 1
        max1 = int (c_left / ow(mm))
        max2 = int (c_left / ow(mm1))
      lar_profit = 0
      best_item1 = 0
      best_item2 = 0
      do 13 i = max1, 0, -1
         do 14 j = 0, max2
           isum = (i * ow(mm) + j * ow(mm1))
             if (isum .gt. c_left) goto 17
                  profit = (i * op(mm)) + (j * op(mm1))
             if (profit .gt. lar_profit) then
                  lar_profit = profit
```

```
               best_item1 = i
               best_item2 = j
            end if
14        continue
         goto 13
17        left = c_left - ow(mm) * (i + 1)
         lprofit = int((left - 1)/((ow(mm1) + 1) * op(mm1)))
         if (lprofit .lt. op(mm)) goto 18
13     continue
18           x(mm) = best_item1
          x(mm1) = best_item2
          c_left = c_left - (x(mm) * ow(mm)) - (x(mm1) * ow(mm1))
          store = store + x(mm) * op(mm) + x(mm1) * op(mm1)
          if (c_left .lt. ismall) goto 15
12     continue
15     write (7, 30)
20     format ( '  extended greedy heuristic  ;',
     +'  n (undominated) =' i6, ';    ', 'c = ' i8)
30     format ('       item     number    weight   profit   ratio')
      do 16 ii = 1, m
      if (x(ii) .lt. 1) goto 16
      write (7, '(2x, 4i10, 2x, f7.3)')  ii, x(ii), ow(ii), op(ii),
     +oratio(ii)
16     continue
      write (*, *) store
      call fdate (string)
      write (*, *), ' finished time : ' , string
      write (7, 40)
40     format ( '  z(h2) =' )
      write (7, *) store
      write (7, 50)
50     format ( 53h          ---------------------------------         )
      do 90 jj = 1, m
        ow(jj) = w(m)
        op(jj) = p(m)
        x(jj) = 0
90     continue
      return
      end
```

TOT_VAL (Total Value Heuristic)

```
c   this program solves the unbounded knapsack problem
c   by the algorithm of white (see ejor, 62(1992), pp.85-95)
```

```
c     n = number of items
c     ic = initial capacity of knapsack
c     p(j) = value of jth item
c     w(j) = weight of jth item
c     t = total value of the solution
c     c_left = remaining capacity of knapsack
c     iaa(j) = number of item j in the knapsack
c     icheck = a check for elimination of dominated items
c     if icheck = 1 or more, the input data will go through a stage of
c     elimination of dominated items
        integer t, w(50000), p(50000), iaa(50000), ind(50000)
        integer iflag(50000)
        real ratio(50000)
        integer low_c, high_c, incre
        character * 15 dfname
      character * 15 dfname2
      character * 30 string
        write (*, *) 'input the number of items, n:'
        read *, n
        print*, 'input the capacity range low-c, high-c of the knapsack: '
        read *, low_c, high_c
        print *, 'input the increment for the knapsack capacity: '
        read *, incre
        write (*, *) 'write the name of the datafile:  '
        read (*, 10) dfname
10    format (a15)
        if (n .gt. 50000) stop 01
        open (unit = 7, file = dfname)
      write (*, *) 'write the name of the output file :'
      read (*, 2) dfname2
      open (unit = 9, file = dfname2)
2    format (a15)
        do 21 j = 1, n
      rewind unit =7
      read (7, '(f7.3, 2i12)') ratio(j), w(j), p(j)
21     continue
        close (unit = 7)
        write (*, *) 'do you like to check for dominance? 1(yes), 0(no)'
        read *, icheck
        if (icheck .ge. 1) call domcheck (n, w, p)
        call fdate (string)
        write (*, *) ' starting time :', string
        do 11 ic = low_c, high_c, incre
        c_left  = ic
        write (9, 101) n , ic
```

```
101   format ( '  total value heuristic ;', '  n (undominated) =' i6,
    +';    ', 'c = ' i8)
        do 22 j = 1, n
22    iflag (j) = 1
        do 23 j = 1, n
23    iaa(j) = 0
        do 24 j = 1, n
24    ind(j) = 0
        t = 0
51    maxind = 0
        ienter = 0
        do 25 j = 1, n
        if (iflag(j) .lt. 0) goto 25
        ind(j) = p(j) * int(c_left / w(j))
        if (maxind .lt. ind(j)) then
          maxind = ind(j)
          ienter = j
        end if
25    continue
        if (ienter .le. 0) goto 52
        iaa(ienter) = int(c_left / w(ienter))
        iflag(ienter) = -9
        c_left = c_left - (iaa(ienter) * w(ienter))
        t = t + (p(ienter) * iaa(ienter))
        write (*, *) t
      call fdate (string)
      write (*, *) ' finished time : ', string
        goto 51
52    write (9, 102)
102   format ('       item      number    weight   profit   ratio')
        do 26 j = 1, n
        if (iaa(j) .lt. 1) goto 26
        write (9, '(2x,4i10,2x,f7.3)') j, iaa(j), w(j), p(j), ratio(j)
26      continue
        write (9, 103) t
103   format (  '  z(tv) = ', x, i10)
        write (9, 104)
104   format ( 53h        -------------------------------          )
11    continue
        goto 99
99    stop
      end
```

CTVG (Complementary Total-Value Greedy Heuristic)

```
c    n = number of items
c    ic = initial capacity of knapsack
c    p(j) = value of jth item
c    w(j) = weight of jth item
c    tv1, tv2 = total value of the solution
c    c_left = remaining capacity of knapsack
c    iaa(j) = number of item j in the knapsack
c    icheck = a check for elimination of dominated items
c    if icheck = 1 or more, the input data will go through a stage of
c    elimination of dominated items
c    ismall = smallest item weight
        integer  w(10000), p(10000), iaa(10000), num(10000)
        integer iflag(10000)
        real ratio(10000)
        integer low_c, high_c, incre
        character * 15 dfname1, dfname2
        character * 11 result
        write (*, *) 'input the number of items, n:'
        read *, n
        print*, 'input the capacity range low-c, high-c of the knapsack: '
        read *, low_c, high_c
        print *, 'input the increment for the knapsack capacity: '
        read *, incre
        write (*, *) 'write the name of the datafile to be sorted:  '
        read (*, 10) dfname1
 10   format (a15)
        if (n .gt. 10000) stop 01
        open (unit = 4, file = dfname1)
        do 11 j = 1, n
 11    read (4, '(f7.3, 2i12)', err=91) (ratio(j), w(j), p(j))
        close (unit = 4, file = dfname1)
        call densort (n, ratio, w, p)
        write (*, *) 'write the name of the sorted datafile:  '
        read (*, 12) dfname2
 12   format (a15)
        open (unit = 5, file = dfname2)
        open (6, file = 'ctot_val.out')
        do 21 j = 1, n
 21   read (5, '(f7.3, 2i12)', err=91) (ratio(j), w(j), p(j))
        close (unit = 5, file = dfname2)
        write (*, *) 'do you like to check for dominance? 1(yes), 0(no)'
        read *, icheck
        if (icheck .ge. 1) call domcheck (n, w, p, ismall)
```

```
       call time (result)
       write (*, *) ' starting time :', result
    do 13 ic = low_c, high_c, incre
       c_left = ic
       write (6, 101) n , c_left
       write (6, 102)
       do 22 j = 1, n
22   iflag (j) = 1
       do 23 j = 1, n
23   iaa(j) = 0
       do 24 j = 1, n
24   num(j) = 0
       ctotval = 0
51   large = 0
    seclarge = 0
    ienter1 = 0
    ienter2 = 0
       t = 0
       tv1 = 0
       tv2 = 0
       do 25 j = 1, n
       if (iflag(j) .lt. 0) goto 25
       num(j) = p(j) * int(c_left / w(j))
       if ((num(j) .ge. large) .and. (num(j) .ge. seclarge)) then
        if (large .ge. seclarge) then
         seclarge = large
         large = num(j)
         ienter2 = ienter1
         ienter1 = j
        else
         large = num(j)
         ienter1 = j
        end if
       else if ((num(j) .le. large) .and. (num(j) .ge. seclarge)) then
         seclarge = num(j)
         ienter2 = j
       end if
25   continue
       if ((ienter1 .eq. 0) .and. (ienter2 .eq. 0)) goto 52
       iaa(ienter1) = int(c_left / w(ienter1))
       iaa(ienter2) = int(c_left / w(ienter2))
       iflag(ienter1) = -9
       iflag(ienter2) = -9
       tv1 = tv1 + (p(ienter1) * iaa(ienter1))
       tv2 = tv2 + (p(ienter2) * iaa(ienter2))
```

```
          if (ienter1 .eq. 1) then
           t = tv1
           write (*, *) t
           c_left = c_left - (iaa(ienter1) * w(ienter1))
           write (6, '(2x,4i10,2x,f7.3)') ienter1, iaa(ienter1),
     +    w(ienter1), p(ienter1), ratio(ienter1)
          else if (ienter2 .eq. 1) then
            t = tv2
            write (*, *) t
            c_left = c_left - (iaa(ienter2) * w(ienter2))
           write (6, '(2x,4i10,2x,f7.3)') ienter2, iaa(ienter2),
     +    w(ienter2), p(ienter2), ratio(ienter2)
           else
            t = tv1
            write (*, *) t
            c_left = c_left - (iaa(ienter1) * w(ienter1))
           write (6, '(2x,4i10,2x,f7.3)') ienter1, iaa(ienter1),
     +    w(ienter1), p(ienter1), ratio(ienter1)
          end if
          ctotval = ctotval + t
          call time (result)
          write (*, *) ' finished time : ', result
          if (c_left .lt. ismall) goto 52
          goto 51
52      write (6, 103) ctotval
101    format ( '  complementary total value heuristic ;',
     +'  n (undominated) =' i5, ';', 'c = ' i7)
102    format ('      item     number    weight   profit   ratio')
103    format (  '  z(ctvg) = ', x, i10)
          write (6, 104)
104    format ( 53h        ------------------------------           )
13     continue
          goto 99
91     write (*, *) '  please check your datafile and rerun'
99     stop
          end
```

Dominance Check Subroutine

```
        subroutine domcheck (n, w, p)
c       this program eliminates the dominated items
          integer w(50000), p(50000), dw(50000), dp(50000)
          open (8, file = 'domi.out')
          do 1 j = 1, n
           dw(j) = w(j)
           dp(j) = p(j)
1     continue
          write (8, *)
c     dominance tests
          do 2 i = 1, n - 1
          if (dw(i) .le. 0) goto 2
            do 3 j = i + 1, n
              if (dw(j) .le. 0) goto 3
              if (int(dw(i) / dw(j)) * dp(j) .ge. dp(i)) then
                      dw(i) = -9
                      go to 2
              end if
              if (int(dw(j) / dw(i)) * dp(i) .ge. dp(j)) then
                      dw(j) = -9
                      go to 3
              end if
3       continue
2      continue
          k = 0
          do 4 i = 1, n
            if (dw(i) .le. 0) goto 4
            k = k + 1
            w(k) = dw(i)
            p(k) = dp(i)
4      continue
          write (8, *) '  data (after dominance test)'
          write (8, 106)
106   format (8x, 'no.', 9x, 'w', 9x, 'p')
          do 5 i = 1, k
5      write (8, 105) i, w(i), p(i)
          write (8, 107) n, k
          n = k
          return
104   format ('  no. of items:', i6, '     number',
   +'      w        p')
105   format (3(i11))
107   format ('  out of ', i6, ' items ', i6, '  are undominated')
          end
```

Appendix C : Analysis of Different Ratio Ranges in the Generation of Class IV Problems

The performance of the density-ordered greedy heuristic (H_1), the extended greedy heuristic (H_2) and the total-value greedy heuristic (TV) for the Class IV problems with varying ratio ranges is discussed in this Appendix. The data sets for this class was randomly generated with w_j uniformly random in [1, 9999] and p_j uniformly random in [1, 9999] such that the ratio p_j / w_j lies in different ratio ranges. The ratio ranges considered are [2.0, 2.5], [3.0, 3.3], [5.0, 5.2], [7.0, 7.1], [11.7, 12.0], [13.1, 13.3] and [16.0, 16.1]. A set of 665 (5 problem instances in the 7 different ratio ranges and 19 knapsack capacities, C $\in$ [10000, 100000] with an increment of 5000 units) test problems with a problem size of 10000 items was randomly generated. The dominance behaviour as shown in Table C1 and the performance of H_1, H_2 and TV with respect to the optimal solution value are analysed.

Table C1: Average number of undominated items for different ratio ranges: Total number of items considered is 10000

Ratio Range	Number of Undominated Items, N
[2.0, 2.5]	200.4
[3.0, 3.3]	365.4
[5.0, 5.2]	484.8
[7.0, 7.1]	556.2
[11.7, 12.0]	272.8
[13.1, 13.3]	398.8
[16.0, 16.1]	312.4

We see form Table C1 that with smaller ratio and narrow range, a large number of undominated items are found. As the ratio range is increased and so is the width in the range, only a few undominated items can be found.

For the ratio range of [2.0, 2.5], [3.0, 3.3], [5.0, 5.2] and [7.0, 7.1], H_1 and H_2 comes close to optimal as the capacity of the knapsack increases and TV performs poorly. For the ratio range of [11.7, 12.0], [13.1, 13.3] and [16.0, 16.1], TV performs better than H_1 and H_2 and in fact TV gives the optimal solution value for all the knapsack capacities. Thus, it can be concluded that for smaller ratio range number and smaller difference in the range, H_1 and H_2 performs better than TV with respect to the optimal solution value and for bigger ratio range number and smaller ratio range difference, TV outperforms H_1 and H_2 and gives the optimal solution value. Figures 1 and 2, for example, show the performance of the three heuritics.

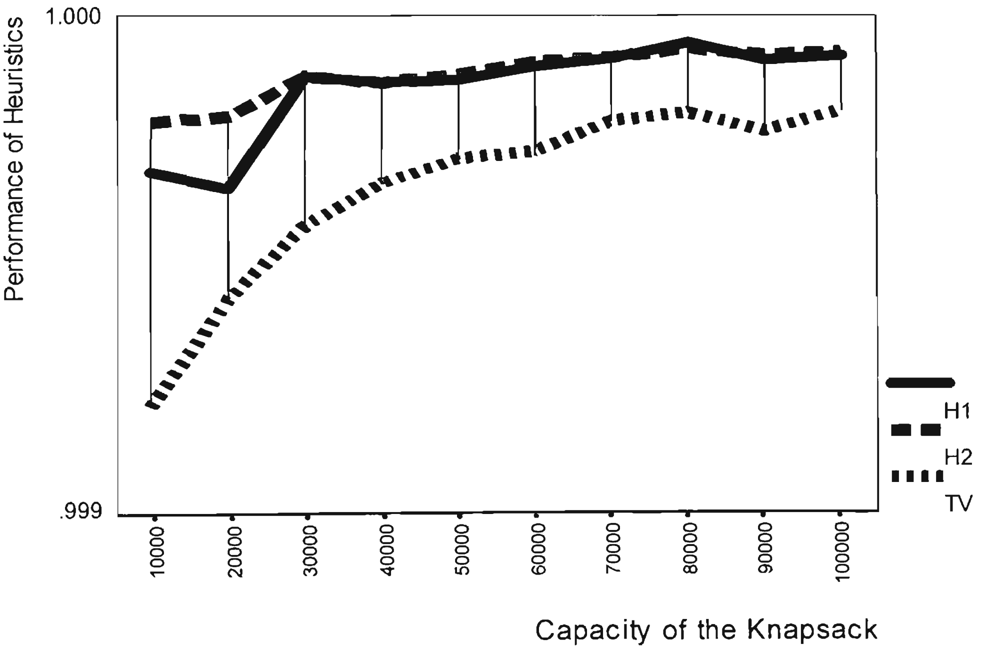

Figure C1: Ratio range [5.0, 5.2] †

† H1 = 0.999685 refers to the case where the heuristic solution is only 0.000315 units away from the optimal. A value close to 1 means the heuristic is close to optimal solution value.

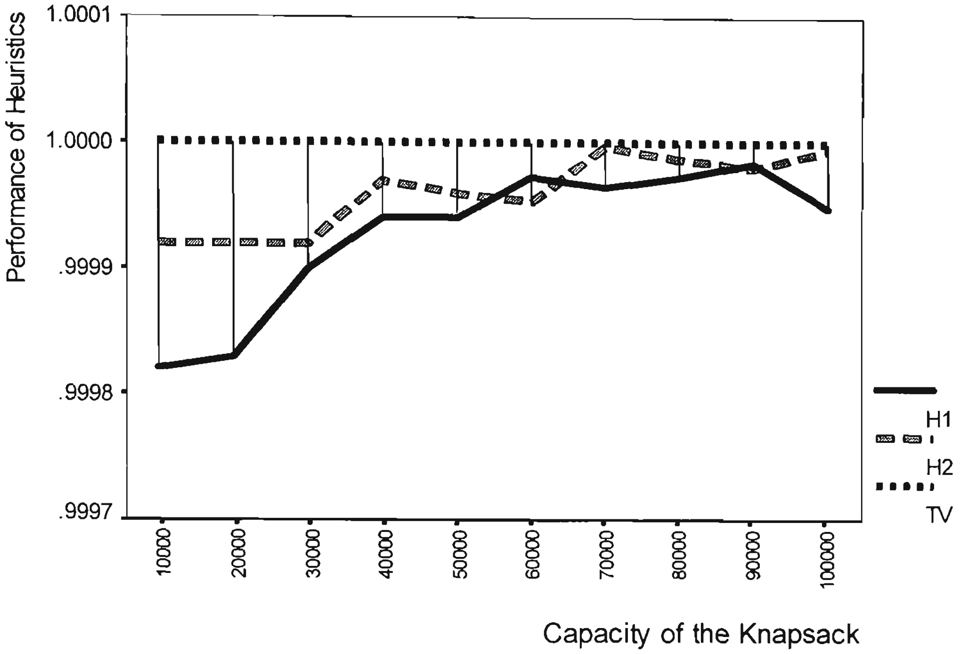

Figure C2: Ratio range [11.7, 12.0]

Appendix D : Difficult Instances of Unbounded Knapsack Problem

The exact algorithm of Martello-Toth (1990) is a very efficient algorithm. But it has been found that for some instances, it is not very efficient. Some such instances are discussed in this Appendix.

In our study, we have found the Martello-Toth algorithm very efficient for the first four problem classes (*Class I* to *IV*). The role of dominance seems to be very crucial.

The fifth class, the *very very strongly correlated* knapsack problem is generated with a high degree of correlation between the item weights and item profits.

Here, w_j is uniformly random in [1, 999]

$$p_j = w_j * \{(w_j - \min w_j + 1) / (\max w_j - \min w_j + 1)\} * 100$$

The idea behind generating problem instances with stronger correlation, is to increase the expected difficulty of corresponding problems, because of an increased number of undominated items.

The UKP is solved by Martello-Toth exact algorithm for different problem sizes (n = 50, 100, 200, 300, 400, 500, 1000, 5000, 10000, 20000, 30000, 40000, 50000) and 10 knapsack capacities C, in the range [100000, 1000000] with an increment of 100000.

For low n ($n \leq 300$) and for high n ($n \geq 1000$), the time to solve optimally by Martello-Toth algorithm was reasonable. The greatest difficulty in terms of computational time arose for problem instances with problem size n = 500.

For one particular instance with capacity 400000 units, Martello-Toth exact algorithm took 55 hours and 36 minutes whereas, the three heuristic algorithms gave near optimal and sometimes optimal solutions in negligible time. This time requirement was consistent across different problem instances with n = 500, although in a few cases, the optimal solution was obtained within hours, or even within a few seconds. Running problems with n = 400 and n = 600 also showed similar time requirements, but the peak appears to be at n = 500.

Table D1: Computational results for *n* = 100, 200, 300, 400 and 500.

Knapsack Capacity	† n = 100 N = 97 S = 0 D = 0.12				n = 200 N = 179 S = 0 D = 0.2				n = 300 N = 261 S = 0 D = 0.24				n = 400 N = 331 S = 0 D = 0.4				n = 500 N = 399 S = 0 D = 0.4			
	H_1	H_2	TV	MTU2 (sec)	H_1	H_2	TV	MTU2 (sec)	H_1	H_2	TV	MTU2 (sec)	H_1	H_2	TV	MTU2 (sec)	H_1	H_2	TV	MTU2 (sec)
100000	‡ 0 (14)	0 (14)	0 (14)	0.24	0 (0)	0 (0)	0 (1100)	13.4	0.2 (0)	0 (0)	0 (1100)	61.4	0 (0)	0 (0)	0 (429)	326	0.4 (1)	0.2 (1)	0 (430)	5244.8
200000	0 (3)	0 (3)	0 (4)	0.24	0 (0)	0.2 (0)	0 (0)	4	0 (0)	0 (0)	0 (0)	6356.6	0 (0)	0 (0)	0 (0)	11299.6	0 (0)	0 (0)	0 (0)	21009.4
300000	0 (24)	0.12 (17)	0 (24)	0.12	0.2 (14)	0 (14)	0 (14)	8.2	0 (5)	0 (5)	0 (5)	265.4	0 (5)	0 (5)	0 (5)	660.8	0 (5)	0 (5)	0 (5)	4945
400000	0 (0)	0.12 (46)	0 (203)	0	0 (0)	0 (0)	0 (0)	15.2	0 (1)	0 (1)	0 (276)	916.6	0 (1)	0 (1)	0 (1)	4214.8	0.2 (0)	0 (0)	0 (202)	200131.6
500000	0 (2)	0.12 (164)	0 (1)	0	0 (12)	0 (12)	0 (12)	6.8	0 (2)	0 (2)	0 (2)	15.6	0 (0)	0 (0)	0 (0)	3.4	0 (0)	0 (0)	0 (0)	2.8
600000	0 (11)	0.48 (91)	0 (11)	0.12	0 (2)	0 (2)	0 (2)	1	0.2 (1)	0 (1)	0 (1)	497.8	0 (0)	0 (0)	0 (0)	12.6	0.2 (0)	0 (0)	0 (0)	73.6
700000	0 (5)	0.48 (0)	0 (5)	0.24	0 (0)	0 (0)	0 (0)	21.2	0.2 (0)	0 (0)	0 (0)	7346.4	0 (0)	0 (0)	0 (0)	13585.6	0.2 (0)	0 (0)	0 (0)	24520.6
800000	0 (0)	0.72 (90)	0 (1)	0.24	0 (3)	0 (3)	0 (3)	0	0 (4)	0 (4)	0 (4)	238.6	0 (0)	0 (0)	0 (0)	601.8	0 (1)	0 (1)	0 (1)	4483.4
900000	0.12 (5)	0.84 (87)	0 (5)	0	0.2 (3)	0 (3)	0 (3)	14.8	0 (2)	0.2 (2)	0 (2)	168	0 (1)	0 (1)	0 (1)	255.8	0 (0)	0 (0)	0 (0)	4150.2
1000000	0 (0)	1.32 (0)	0 (0)	0	0 (0)	0 (0)	0 (0)	5.2	0 (0)	0 (0)	0 (0)	35.8	0 (0)	0 (0)	0 (0)	45.6	0 (0)	0 (0)	0 (0)	355.6

† n = Total number of items
N = Number of undominated items
S = time taken for sorting in secs
D = time taken for Dominance test in secs

‡ t = total CPU-seconds
(r) = coded value of heuristic/optimal

Ex., r = 14 means the solution is 0.000014% smaller than optimal

The average running times of the FORTRAN 77 implementations of algorithms DGREEDY, EXTGREED, TOT_VAL and MTU2 computed over 50 problem instances (expressed in seconds) are shown. Sorting times and time taken for the dominance check are also shown separately for each value of n. These times are not included in the running time for heuristic algorithms but are included in the Martello-Toth exact algorithm, MTU2. The reason for such huge running time is not clear for these difficult problem instances and needs further investigation. However, a sample data set is included for reference.

As shown in Table D1, the heuristics H_1, H_2 and TV are much faster than MTU2 and also gives optimal solution value in most instances.

The following is a sample data set of *Class V* with 500 items.

Ratio	*wj*	*pj*
17.918	179	3207
20.120	201	4044
11.512	115	1323
26.226	262	6871
43.143	431	18594
45.445	454	20632
43.744	437	19116
40.841	408	16663
76.777	767	58887
45.145	451	20360
42.442	424	17995
13.714	137	1878
18.018	180	3243
51.952	519	26963
69.570	695	48350
6.206	62	384
58.759	587	34491
20.521	205	4206
27.928	279	7791
97.898	978	95744
93.994	939	88260
79.680	796	63425
24.324	243	5910
11.612	116	1346
89.590	895	80182
74.975	749	56156
31.331	313	9806
37.237	372	13852
62.763	627	39352
4.805	48	230
78.078	780	60900
40.641	406	16500
39.039	390	15225
21.221	212	4498
73.774	737	54371
63.063	630	39729
47.447	474	22490
33.834	338	11435
75.976	759	57665
13.714	137	1878
70.170	701	49189
26.326	263	6923
13.914	139	1934
69.870	698	48769
92.693	926	85833
80.280	802	64384
29.630	296	8770
91.792	917	84173
12.713	127	1614
61.662	616	37983
44.645	446	19911
92.492	924	85463
6.707	67	449
60.060	600	36036
52.753	527	27800
88.088	880	77517
8.408	84	706
42.543	425	18080
22.122	221	4888
93.393	933	87136
71.572	715	51173
86.386	863	74551
91.091	910	82892
65.265	652	42552
91.592	915	83806
31.832	318	10122
64.665	646	41773
95.896	958	91868
98.298	982	96528
27.728	277	7680
98.098	980	96136
73.974	739	54666
77.477	774	59967
7.608	76	578
68.068	680	46286

Ratio	*wj*	*pj*	Ratio	*wj*	*pj*	Ratio	*wj*	*pj*
77.678	776	60277	89.089	890	79289	49.449	494	24428
77.377	773	59812	73.373	733	53782	0.400	4	1
96.897	968	93796	8.809	88	775	85.385	853	72833
52.052	520	27067	85.285	852	72663	59.459	594	35318
30.731	307	9434	95.996	959	92060	90.691	906	82165
5.405	54	291	50.150	501	25125	79.680	796	63425
93.894	938	88072	13.614	136	1851	34.134	341	11639
26.927	269	7243	48.348	483	23352	84.184	841	70798
73.173	731	53489	60.661	606	36760	39.740	397	15776
20.821	208	4330	2.102	21	44	22.723	227	5158
19.219	192	3690	64.264	642	41257	20.220	202	4084
30.931	309	9557	84.685	846	71643	12.713	127	1614
42.543	425	18080	12.212	122	1489	90.691	906	82165
98.498	984	96922	94.695	946	89581	59.860	598	35796
52.152	521	27171	73.574	735	54076	50.450	504	25427
68.468	684	46832	22.723	227	5158	83.884	838	70294
65.465	654	42814	52.052	520	27067	91.892	918	84356
14.014	140	1961	17.618	176	3100	76.677	766	58734
59.560	595	35437	19.219	192	3690	49.249	492	24230
28.829	288	8302	3.003	30	90	76.276	762	58122
34.735	347	12052	62.863	628	39477	20.320	203	4125
36.236	362	13117	65.766	657	43208	79.980	799	63904
48.749	487	23740	9.810	98	961	29.229	292	8534
55.556	555	30833	3.804	38	144	80.180	801	64224
94.494	944	89202	32.733	327	10703	76.376	763	58275
12.913	129	1665	11.111	111	1233	77.177	771	59503
31.231	312	9744	16.416	164	2692	84.484	844	71304
5.105	51	260	15.415	154	2373	29.830	298	8889
83.183	831	69125	85.586	855	73175	48.749	487	23740
96.997	969	93990	40.641	406	16500	80.781	807	65190
54.755	547	29950	71.772	717	51460	21.221	212	4498
64.064	640	41001	10.110	101	1021	50.250	502	25225
30.531	305	9311	13.614	136	1851	64.965	649	42162
87.588	875	76639	72.172	721	52036	75.976	759	57665
65.265	652	42552	73.373	733	53782	18.919	189	3575
59.359	593	35200	23.624	236	5575	52.553	525	27590
92.192	921	84909	35.235	352	12402	86.086	860	74034
3.604	36	129	85.385	853	72833	15.516	155	2404
77.878	778	60588	67.568	675	45608	92.292	922	85093
71.672	716	51316	34.835	348	12122	73.173	731	53489
32.533	325	10573	33.834	338	11435	50.851	508	25832
7.508	75	563	64.665	646	41773	21.121	211	4456
57.758	577	33326	65.465	654	42814	86.887	868	75417
37.437	374	14001	4.304	43	185	74.474	744	55409
29.830	298	8889	13.413	134	1797	36.336	363	13190
45.846	458	20997	21.121	211	4456	59.259	592	35081
69.870	698	48769	68.268	682	46558	70.671	706	49893
8.108	81	656	70.270	702	49329	32.232	322	10378

Ratio	*wj*	*pj*
53.754	537	28865
3.604	36	129
77.077	770	59349
72.773	727	52905
22.623	226	5112
0.901	9	8
81.481	814	66325
50.050	500	25025
89.590	895	80182
53.353	533	28437
40.240	402	16176
90.490	904	81803
36.837	368	13555
39.840	398	15856
48.949	489	23936
34.234	342	11708
40.140	401	16096
44.845	448	20090
30.230	302	9129
25.225	252	6356
28.729	287	8245
19.520	195	3806
74.274	742	55111
79.880	798	63744
91.091	910	82892
53.253	532	28330
94.895	948	89960
12.212	122	1489
34.334	343	11776
13.514	135	1824
47.748	477	22775
24.424	244	5959
84.885	848	71982
50.951	509	25934
16.817	168	2825
23.423	234	5481
58.258	582	33906
1.001	10	10
20.821	208	4330
24.124	241	5813
70.370	703	49470
39.339	393	15460
31.632	316	9995
49.950	499	24925
65.065	650	42292
14.615	146	2133
4.004	40	160
9.610	96	922

Ratio	*wj*	*pj*
33.033	330	10900
77.878	778	60588
75.976	759	57665
3.303	33	109
83.784	837	70127
45.245	452	20450
57.357	573	32865
88.188	881	77693
94.695	946	89581
0.701	7	4
70.671	706	49893
13.514	135	1824
25.125	251	6306
71.271	712	50745
72.973	729	53197
67.568	675	45608
49.650	496	24626
49.449	494	24428
58.959	589	34726
25.325	253	6407
31.031	310	9619
88.388	883	78046
83.784	837	70127
24.024	240	5765
72.873	728	53051
8.208	82	673
68.969	689	47519
22.222	222	4933
44.545	445	19822
57.558	575	33095
2.703	27	72
60.561	605	36639
34.034	340	11571
3.504	35	122
29.029	290	8418
26.527	265	7029
59.159	591	34963
78.078	780	60900
53.754	537	28865
65.966	659	43471
16.817	168	2825
60.761	607	36881
54.054	540	29189
6.106	61	372
44.144	441	19467
7.808	78	609
25.225	252	6356
2.803	28	78

Ratio	*wj*	*pj*
33.734	337	11368
1.201	12	14
28.028	280	7847
79.680	796	63425
79.980	799	63904
23.524	235	5528
65.365	653	42683
18.018	180	3243
45.646	456	20814
45.245	452	20450
48.248	482	23255
67.968	679	46150
77.477	774	59967
9.309	93	865
6.306	63	397
47.447	474	22490
53.453	534	28544
65.666	656	43076
27.928	279	7791
5.205	52	270
15.015	150	2252
40.641	406	16500
12.312	123	1514
45.345	453	20541
75.075	750	56306
76.677	766	58734
7.668	676	45743
64.264	642	41257
23.223	232	5387
74.975	749	56156
6.306	63	397
60.861	608	37003
29.229	292	8534
39.239	392	15381
38.539	385	14837
61.762	617	38107
68.569	685	46969
27.528	275	7570
95.596	955	91293
66.667	666	44400
41.542	415	17239
5.906	59	348
71.071	710	50460
94.695	946	89581
41.942	419	17573
93.994	939	88260
84.685	846	71643
29.930	299	8949

Ratio	*wj*	*pj*
23.223	232	5387
83.984	839	70462
44.044	440	19379
60.160	601	36156
48.348	483	23352
84.985	849	72152
54.154	541	29297
97.097	970	94184
57.057	570	32522
6.006	60	360
23.624	236	5575
43.944	439	19291
40.841	408	16663
41.041	410	16826
34.234	342	11708
48.549	485	23546
77.477	774	59967
23.223	232	5387
49.750	497	24725
73.073	730	53343
16.917	169	2858
78.579	785	61684
76.076	760	57817
46.046	460	21181
12.613	126	1589
67.968	679	46150
87.688	876	76814
9.209	92	847
8.909	89	792
28.529	285	8130
10.310	103	1061
67.768	677	45878
89.890	898	80721
33.634	336	11300
27.427	274	7515
22.122	221	4888
86.386	863	74551
79.379	793	62947
87.487	874	76464
26.026	260	6766
94.595	945	89391
16.717	167	2791
13.714	137	1878
76.877	768	59041
61.662	616	37983
12.212	122	1489
28.929	289	8360
54.254	542	29405
16.617	166	2758
61.161	611	37369
11.712	117	1370
77.277	772	59658
36.637	366	13409
6.406	64	410
87.888	878	77165
47.247	472	22300
52.252	522	27275
52.953	529	28012
86.787	867	75244
8.208	82	673
56.957	569	32408
70.671	706	49893
39.239	392	15381
10.511	105	1103
23.023	230	5295
29.530	295	8711
8.609	86	740
54.254	542	29405
48.248	482	23255
74.174	741	54963
35.636	356	12686
76.777	767	58887
41.942	419	17573
97.598	975	95157
18.819	188	3537
22.122	221	4888
92.993	929	86390
45.546	455	20723
97.097	970	94184
80.380	803	64545
92.793	927	86018
31.732	317	10058
89.289	892	79646
77.377	773	59812
80.480	804	64706
4.304	43	185
95.195	951	90530
2.603	26	67
93.794	937	87884
90.090	900	81081
20.721	207	4289
90.791	907	82347
79.980	799	63904
45.646	456	20814
45.245	452	20450
95.696	956	91485
76.376	763	58275
91.792	917	84173
47.748	477	22775
32.432	324	10508
1.902	19	36
40.941	409	16744
30.831	308	9495
44.044	440	19379
40.040	400	16016
42.943	429	18422
91.992	919	84540
69.169	691	47795
39.139	391	15303
9.910	99	981
6.707	67	449
93.493	934	87322
43.043	430	18508
0.400	4	1
4.605	46	211
68.068	680	46286
77.978	779	60744
21.522	215	4627
1.502	15	22
29.229	292	8534
17.818	178	3171
62.162	621	38602
44.845	448	20090
21.622	216	4670
10.811	108	1167
85.886	858	73690
13.213	132	1744
17.818	178	3171
82.783	827	68461
65.666	656	43076
40.040	400	16016
31.331	313	9806
46.747	467	21830
76.476	764	58428
16.216	162	2627
55.055	550	30280
58.458	584	34139

REFERENCES

Aarts, E.H.L. and J.H.M. Korst (1989), *Simulated Annealing and Boltzmann Machines. A Stochastic Approach to Combinatorial Optimisation and Neural Computing,* Wiley, Chichester.

Abramson, D.A. , H. Dang and M. Krishnamoorthy (1996), A Comparison of Two Methods for Solving 0-1 Integer Programs Using a General Purpose Simulated Annealing Algorithm, *Annals of Operations Research,* Vol.63, 129

Aittoniemi, L. and K. Oehlandt (1985), A Note on the Martello-Toth Algorithm for One-dimensional Knapsack Problems, *European Journal of Operational Research,* Vol.20, 117.

Bahrami, A. and C.H. Dagli (1994), Hybrid Intelligent Packing System (HIPS) Through Integration of Artificial Neural Networks, Artificial-intelligence, and Mathematical-programming, *Applied Intelligence,* Vol.4, 321.

Balas, E. and E. Zemel (1980), An Algorithm for Large 0-1 Knapsack Problems, *Operations Research,* Vol.28, 1130-1154.

Barr, R.S., B.L. Golden, J.P. Kelly, M.G.C. Resende and W.R. Stewart, Jr (1995), Designing and Reporting on Computational Experiments with Heuristic Methods, *Journal of Heuristics,* Vol.1, 9-32.

Bulfin, R.L., R.G. Parker and C.M. Shetty (1979), Computational Results With a Branch and Bound Algorithm for the General Knapsack Problem, *Naval Research Logistics Quarterly,* Vol.26, 41-46.

Cabot, A.V. (1970), An Enumeration Algorithm for Knapsack Problems, *Operations Research,* Vol.18, 306-311.

Cagan, J. (1994), Shape Annealing Solution to the Constrained Geometric Knapsack Problem, *Computer Aided Design,* Vol.26, 763.

Dammeyer, F. and S. Voss (1993), Dynamic Tabu List Management Using the Reverse Elimination Method, *Annals of Operations Research,* Vol.41, 29-46.

Dantzig, G.B. (1957), Discrete Variable Extremum Problems, *Operations Research,* Vol.5, 266-277.

Davis, E.W (1974), Networks: Resource Allocation, *Journal of Industrial Engineering,* Vol.6, 22-32.

De Jong, K. (1975), An Analysis of the Behaviour of a Class of Genetic Adaptive Systems, *Doctoral Dissertation,* University of Michigan, Ann Arbor.

Drexl, A. (1988), A Simulated Annealing Approach to the Multiconstraint Zero-one Knapsack Problem, *Computing,* Vol.40, 1.

Dudzinski, K. (1991), A Note on Dominance Relation in Unbounded Knapsack Problems, *Operations Research Letters,* Vol.10, 417-419.

Fairley, A. and D.F. Yates (1993), An Alternative Method of Choosing the Crossover Point when Solving the Knapsack Problem with Genetic Algorithms, *Working Paper, Department of Computer Science, University of Liverpool.*

Falkenauer, E. and A. Delchambre (1992), A Genetic Algorithm for Bin Packing and Line Balancing, *Proceedings of the 1992 IEEE International*

Conference on Robotics and Automation (IEEE Computer Society Press, Los Alamitos, California), 1186.

Fisher, M.L. (1980), Worst Case Analysis of Heuristic Algorithms, *Management Science,* Vol.26, No.1, 1-17.

Garey, M.R. and D.S. Johnson (1979), *Computers and Intractability,* Freeman, New York.

Garfinkel, R.S. and G.L. Nemhauser (1972), *Integer Programming,* New York, Wiley.

Gilmore, P.C. and R.E. Gomory (1963), A Linear Programming Approach to the Cutting Stock Problem II, *Operations Research,* Vol.11, 863-888.

Gilmore, P.C. and R.E. Gomory (1966), The Theory and Computation of Knapsack Functions, *Operations Research,* Vol.14, 1045-1074.

Glover, F. (1986), Future Paths for Integer Programming and Links to Artificial Intelligence, *Computers and Operations Research,* Vol.1, 533-549.

Glover, F. (1994), Optimisation by Ghost Image Processes in Neural Networks, *Computers and Operations Research,* Vol.21, 801.

Goldberg, D.E. (1989), *Genetic Algorithms in Search, Optimisation, and Machine Learning,* Addison-Wesley, New York.

Hanafi, S., A. Freville and A. El-Abdellaoui (1996), Comparison of Heuristics for the 0-1 Multidimensional Knapsack Problem, *Metaheuristics. Theory and Applications,* ed. I.H. Osman and J.P. Kelly (Kluwer, Boston).

Hansen, P. and J. Ryan (1996), Testing Integer Knapsacks for Feasibility, *European Journal of Operational Research,* Vol.88, 578-582.

Holland, J.H. (1992, 1975), *Adaptation in Natural and Artificial Systems,* 2 ed. MIT Press, Cambridge, (1 ed. 1975, The University of Michigan Press, Ann Arbor).

Hopfield, J.J. and D.W. Tank (1985), Neural Computation of Decisions in Optimisation Problems, *Biological Cybernetics,* Vol.52, 141-152.

Horowitz, E. and S. Sahni (1978), *Fundamentals of Computer Algorithms,* Computer Science Press, Rockville.

Hu, T.C. and M.L. Lenard (1975), Optimality of a Heuristic Solution for a Class of Knapsack Problems, *Operations Research,* Vol.24, 193-196.

Ingargiola, G.P. and J.F. Korsh (1973), A Reduction Algorithm for 0-1 Single Knapsack Problems, *Management Science,* Vol.20, No.4, 460-463.

Ingargiola, G.P. and J.F. Korsh (1977), A General Algorithm for One-Dimensional Knapsack Problems, *Operations Research,* Vol.25, 752-759.

Johnston, R.E. and L.R. Khan (1995), A Note on Dominance in Unbounded Knapsack Problems, *Asia-Pacific Journal of Operational Research,* Vol.12, No.2, 145-160.

Kirkpatrick, S., C.D. Gelatt and P.M. Vecchi (1983), Optimisation by Simulated Annealing, *Science,* Vol.220, 671-680.

Kohli, R. and R. Krishnamurti (1992), A Total-value Greedy Heuristic for the Integer Knapsack Problem, *Operations Research Letters,* Vol.12, 65-71, North-Holland.

Kohli, R. and R. Krishnamurti (1995), Joint Performance of Greedy Heuristics for the Integer Knapsack Problem, *Discrete Applied Mathematics,* Vol.56, 37-48, Elsevier Science.

Kurokawa, T. and S. Kozuka (1994), Use of Neural Networks for the Optimum Frequency Assignment Problem, *Electronics and Communications in Japan Part I - Communications,* Vol.77, 106.

Laguna, M. and F. Glover (1993), Bandwidth Packing. A Tabu Search Approach, *Management Science,* Vol.39, 492.

Lai, T.C. (1993), Worst-case Analysis of Greedy Algorithms for the Unbounded Knapsack, Subset-sum and Partition Problems, *Operations Research Letters,* Vol.14, 215-220, North-Holland.

Lorie, J. and L. Savage (1955), Three Problems in Capital Rationing, *Journal of Business,* Vol.28, 229-239.

Magazine, M.J., G.L. Nemhauser and L.E. Trotter, Jr. (1975), When the Greedy Solution Solves a Class of Knapsack Problems, *Operations Research,* Vol.23, No.2, 207-217.

Martello, S. and P. Toth (1988), A New Algorithm for the 0-1 Knapsack Problem, *Management Science,* Vol.34, No.5, 633-644.

Martello, S. and P. Toth (1990), *Knapsack Problems - Algorithms And Computer Implementations,* Chichester; New York, Wiley.

Metropolis, W., A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller (1953), Equations of the State Calculations by Fast Computing Machines, *Journal of Chemical Physics,* Vol.21, 1087-1092.

Nauss, R.M. (1976), An Efficient Algorithm for the 0-1 Knapsack Problem, *Management Science,* Vol.23, No.1, 27-31.

Nemhauser, G.L. and Z. Ullmann (1969), Discrete Dynamic Programming and Capital Allocation, *Management Science,* Vol.15, No.9, 494-505.

Ohlsson, M., C. Peterson and B. Soderberg (1993), Neural Networks for Optimisation Problems with Inequality Constraints. The Knapsack Problem, *Neural Computation,* Vol.5, 331.

Osman, I.H. and J.P. Kelly (1996), *Meta-Heuristics: Theory and Applications,* Kluwer Academic Publishers, Massachusetts.

Osman, I.H. and G. Laporte (1996), Metaheuristics: A Bibliography, *Annals of Operations Research,* Vol.63, 513-623.

Reeves, C.R. (1993), *Modern Heuristic Techniques for Combinatorial Problems,* Halsted Press, New York.

Reeves, C.R. (1996), Hybrid Genetic Algorithms for Bin-packing and Related Problems, *Annals of Operations Research,* Vol.63, 371.

Salkin, H.M. and C.A. de Kluyver (1975), The Knapsack Problem: A Survey, *Naval Research Logistics Quarterly,* Vol.22, 127-144.

Salkin, H.M. and K. Mathur (1989), *Foundations Of Integer Programming,* New York, North-Holland.

Sahni, S. (1975), Approximate Algorithms for the 0-1 Knapsack Problems, *Journal of the Association for Computing Machinery,* Vol.22, No.1, 115-124.

Taha, H.A. (1975), *Integer Programming Theory, Applications and Computations,* New York, Academic Press.

Thiel, J. and S. Voβ (1994), Some Experiences on Solving Multiconstraint Zero-one Knapsack Problems with Genetic Algorithms, *INFORS,* Vol.32, 226.

White, D.J. (1991), An Extension of a Greedy Heuristic for the Knapsack Problem, *European Journal of Operational Research,* Vol.51, 387-399, North-Holland.

White, D.J. (1992), A Complementary Greedy Heuristic for the Knapsack Problem, *European Journal of Operational Research,* Vol.62, 85-95, North-Holland.

Zhu, N. and K. Broughan (1996), A Note on Reducing the Number of Variables in Integer Programming Problems, *Computational Optimisation and Applications,* Vol.8, No.3, 263-272.

BIBLIOGRAPHY

Ahrens, H.J. and G. Finke (1975), Merging and Sorting Applied to the Zero-One Knapsack Problem, *Operations Research,* Vol.23, No. 6, 1099-1109.

Babayev, D.A. and S.S. Mardanov (1994), Reducing the Number of Variables in Integer and Linear Programming Problems, *Computational Optimisation and Applications,* Vol.3, 99-109.

Chiu, S.Y., L. Lu and L.A. Cox, Jr. (1996), Optimal Access Control for Broadband Services: Stochastic Knapsack with Advance Information, *European Journal of Operational Research,* Vol.89, 127-134.

Horowitz, E. and S. Sahni (1974), Computing Partitions with Applications to the Knapsack Problem, *Journal of the Association for Computing Machinery,* Vol.21, No.2, 277-292.

Ibarra, O.H. and C.E. Kim (1975), Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems, *Journal of the Association for Computing Machinery,* Vol.22, No.4, 463-468.

Kannan, R. And B. Korte (1984), Approximative Combinatorial Algorithms, *Mathematical Programming,* Elsevier Science Publishers B. V. (North-Holland).

Krass, D. and S.P. Sethi (1994), Some Complexity Issues in a Class of Knapsack Problems: What Makes a Knapsack Problem "Hard"?, *INFOR,* Vol.32, No.3, 149-161.

Li, D., H. Lin and K. Torng (1996), A Strategy for Evolution of Algorithms to Increase the Computational Effectiveness of NP-hard Scheduling Problems, *European Journal of Operational Research,* Vol.88, 404-412.

Martello, S. and P. Toth (1977), Branch and Bound Algorithms for the Solution of the General Unidimensional Knapsack Problem, *In M. Roubens (ed.), Advances in Operations Research,* North-Holland, Amsterdam, 295-301.

Martello, S. and P. Toth (1984), A Mixture of Dynamic Programming and Branch-and-Bound for the Subset-Sum Problem, *Management Science,* Vol.30, No.6, 765-771.

Pisinger, D. (1995), Avoiding Anomalies in the MT2 Algorithm by Martello and Toth, *European Journal of Operational Research,* Vol.82, 206-208.

Pisinger, D. (1995), An Expanding-core Algorithm for the Exact 0-1 Knapsack Problem, *European Journal of Operational Research,* Vol.87, 175-187.

Zoltners, A.A. (1978), A Direct Descent Binary Knapsack Algorithm, *Journal of the Association for Computing Machinery,* Vol.25, No.2, 304-311.