APPROXIMATION ALGORITHMS FOR MINIMUM KNAPSACK PROBLEM

MOHAMMAD TAUHIDUL ISLAM
Bachelor of Science, Islamic University of Technology, 2005

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

*I dedicate this thesis to my **parents**.*

Abstract

Knapsack problem has been widely studied in computer science for years. There exist several variants of the problem, with zero-one maximum knapsack in one dimension being the simplest one. In this thesis we study several existing approximation algorithms for the minimization version of the problem and propose a scaling based fully polynomial time approximation scheme for the minimum knapsack problem. We compare the performance of this algorithm with existing algorithms. Our experiments show that, the proposed algorithm runs fast and has a good performance ratio in practice. We also conduct extensive experiments on the data provided by Canadian Pacific Logistics Solutions during the MITACS internship program.

We propose a scaling based ε-approximation scheme for the multidimensional (d-dimensional) minimum knapsack problem and compare its performance with a generalization of a greedy algorithm for minimum knapsack in d dimensions. Our experiments show that the ε-approximation scheme exhibits good performance ratio in practice.

Acknowledgments

I thank my supervisor, Dr. Daya Gaur, for many insightful discussions that help develop the ideas in the thesis. I express deep gratitude to him for being the source of a constant motivation and inspiration throughout my Masters degree.

I would like to thank my M.Sc. supervisory committee members Dr. Hadi Kharaghani, Dr. Shahadat Hossain and Dr. Paul Hazendonk for their useful suggestions and propositions. I thank my external examiner Dr. Asish Mukhopadhyay for his highly perceptive comments and suggestions.

For the financial support I would like to thank Dr. Daya Gaur and the School of Graduate Studies, University of Lethbridge.

I would like to thank MITACS (Mathematics of Information Technology and Complex Systems) for providing me the internship opportunity at CPLS (Canadian Pacific Logistics Solutions) where I learned a lot of things related to working in a large organization that I could have never learned if I had not availed myself of the opportunity.

I express my deep gratitude to my parents and my wife without whom it would not have been possible to complete this thesis. I would also like to thank my fellow graduate students who encouraged me throughout the program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the most well-known problems in computer science is the Knapsack problem. It is one of the problems on Karp's original list of 21 *NP*-complete problems [21]. This problem is a well-known decision problem where the feasibility of a particular selection of alternatives can be evaluated by a linear combination of coefficients for each binary decision. A binary decision can include an item or exclude the item from selection. The feasibility of an alternative is determined by one or more capacity constraints. As the name implies the problem can be viewed as a decision problem faced by a person who wants to pack his or her knapsack of capacity C with a subset of items from the item set $S = \{1, ..., n\}$ such that the total profit of items packed in the knapsack is maximized and the total weight of the selected items does not exceed the knapsack capacity C. From the context it is clear that, each item i will have a profit p_i and a weight w_i associated with it.

A Maximum knapsack instance is given as a set of n items $\{1, ..., n\}$ each having a profit p_i and weight w_i associated with it and a capacity value C associated with the knapsack. The objective is to select the most profitable subset of items such that the total weight of the selected items is at most the capacity value C.

The maximum knapsack problem can be formulated as an integer linear program as follows:

$$\begin{aligned}
&\text{maximize } \sum_{i=1}^{n} p_i x_i \\
&\text{subject to } \sum_{i=1}^{n} w_i x_i \leq C \\
&\qquad x_i \in \{0,1\}, \forall \quad i \in \{1,...,n\}
\end{aligned}$$

If $x_i = 1$ then item i is included in the knapsack and $x_i = 0$ indicates that the item is not included in the knapsack. This is one of the simplest integer programs with only one linear constraint over n binary variables. Because of its simplicity it has been studied extensively as a prototype for maximization problems. However no polynomial time algorithm is known for solving the above integer program (in the worst case).

The objective of the minimum knapsack problem is to find the least profitable set of items such that the total weight of the selected items is at least the capacity C. The minimum-knapsack problem can be formulated as the following integer program:

$$\begin{aligned}
&\text{minimize} \sum_{i=1}^{n} p_i x_i \\
&\text{subject to} \sum_{i=1}^{n} w_i x_i \geq C \\
&\qquad x_i \in \{0,1\}, \forall \quad i \in \{1,...,n\}
\end{aligned}$$

1.1 Real world importance of the knapsack problem

The knapsack problem is not only of theoretical interest but it can also be used to model several real world scenarios. If we consider an investor who wants to invest in several

projects to maximize profit on his investments but has a fixed monetary capacity of C. Each of the available projects has a cost associated with it and also a profit value that the investor can benefit from. The goal of the investor is to invest money into a subset of the projects so as not to exceed the monetary capacity C and maximize the profit in the process.

A second example can be given for the minimum knapsack problem. Consider a factory that needs D units of some raw material. There are n places from where one unit of the raw material can be procured at cost c_j. The goal is to meet the demand at minimum cost. The example can be generalized to capture a more realistic scenario.

1.2 Input specifications for the maximum knapsack problem

To avoid some trivial cases we need some assumptions on the input instance of the knapsack problem. We assume that the total number of items n in the input instance is strictly greater than one.

Another assumption is that the weight of every item in the instance is less than or equal to the capacity of the knapsack C. If an item has weight $w_i > C$ then this item can never be included in the knapsack. So without loss of generality we assume that

$$w_i \leq C, \forall i \in \{1, ..., n\}$$

We also assume that, the sum of weights of all the items are greater than the knapsack capacity. Otherwise all items will fit into the knapsack. So we assume that

$$\sum_{i=1}^{n} w_i > C$$

Another assumption is that all the profits p_i and weights w_i are positive and greater than 0, i.e. ,

$$p_i > 0,\ w_i > 0,\ \forall i \in \{1,...,n\}$$

1.3 Variants of the max-knapsack problem

Several variants of the classical knapsack problem have been examined extensively. Some variants are based on the number of copies of an item and other variants are based on the number of constraints involved. Below we provide a brief description on the variants of the knapsack problem. For details please see the book by Martello and Toth [34]. Note that the following discussion applies to both maximum and minimum knapsack problems.

1.3.1 Variants based on copies of items

According to this scheme knapsack problems can be divided into two classes as follows:

Bounded knapsack problem

The number of times an item j can be copied into the knapsack is bounded by some positive integer b_j. The constraint in the integer program then becomes,

$$0 \leq x_j \leq b_j, \quad x_j \text{ is an integer} \qquad \forall j \in \{1, ..., n\}$$

Here x_j is the decision variable for item j. In the case of $0/1$ knapsack problem b_j is 1 for each item j.

Unbounded knapsack problem

Another variant is the *unbounded knapsack problem*. It is also known as the *integer knapsack problem*. An item can be copied integrally many times. In this case the constraint becomes,

$$x_j \geq 0, \quad x_j \text{ is an integer}, \qquad \forall j \in \{1, ..., n\}.$$

Here n is the number of items in the knapsack instance.

1.3.2 Variants based on the number of constraints

We also classify the knapsack problems based on the number of constraints involved. If the problem formulation has only one constraint involved then it is called the *one dimensional knapsack problem* or generally *knapsack problem*. And if the problem formulation contains $d \geq 1$ constraints then it is called *d-dimensional knapsack* or *multidimensional knapsack*. In the multidimensional knapsack problem formulation the constraint,

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

is replaced by d constraints,

$$\sum_{j=1}^{n} w_{ij} x_j \leq C_i, \qquad \forall \quad i \in \{1, ..., d\}$$

In the d-dimensional version of the problem, each item can have a possibly different weight in each dimension i for all $i \in \{1, ..., d\}$, and the goal is to select a subset of items with maximum profit that satisfy the capacity constraint in each of the d dimensions. Note that the minimum knapsack problems can be classified similarly.

1.4 Computational Complexity

Now we introduce some complexity theoretic definitions from Garey and Johnson [15].

1.4.1 Class *P* and *NP*

Class P is the set of decision problems that can be solved in polynomial time on a deterministic Turing machine [15]. Class NP is the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine. Graph coloring, vertex cover, travelling salesman and graph isomorphism problems belong to the class NP. An equivalent definition of class NP is the set of problem for which there exist polynomial time "verifiers". The simplest definition of the graph coloring problem is to assign least number of colors to the nodes in the graph such that no two adjacent vertices share the same color. Once colors are assigned to the nodes it is easy to verify that whether two adjacent vertices share the same color or not.

1.4.2 Polynomial-time reductions

If each instance of problem Y can be transformed to an instance of problem X in polynomial time in the size of the instance such that all the satisfiable instances of problem Y map to the satisfiable instances of problem X and vice-versa, then we say that, Y is polynomially reducible to X, written as $Y \leq_P X$. In other words, X is at least as "hard" as Y, as existence of a polynomial time algorithm for X implies that Y can be solved in polynomial time.

1.4.3 NP-Complete problems

A problem X is called NP-complete if the following two properties hold for X:

- $X \in NP$.
- For all $Y \in NP$, $Y \leq_P X$.

In other words, the problem is in the class NP and, all the other problems in class NP are polynomially reducible to this problem.

A widely believed conjecture is that, NP-complete problems do not admit a polynomial-time algorithm. Approximation algorithms provide a way of coping with NP-completeness. Approximation algorithms trade running time for closeness to the optimal solution and we discuss approximation algorithms next.

1.4.4 Approximation algorithms

Approximation algorithms can be divided into following types [25]:

- Absolute approximation algorithms.
- k-factor approximation algorithms.

Absolute approximation algorithms

For a minimization optimization problem P an absolute approximation algorithm is a polynomial time algorithm A such that, there exists a constant k and for all instances I of P,

$$|A(I) - OPT(I)| \leq k,$$

where $A(I)$ is the value of the solution returned by algorithm A on instance I and $OPT(I)$ is the value of the optimal solution for the instance I. An absolute approximation algorithm is very rare in practice. For a very few NP-complete optimization problems an absolute approximation algorithm is known. Absolute approximation algorithms exist for the edge coloring problem and the vertex coloring problem for planar graphs [25].

k-factor approximation algorithms

For a maximization optimization problem P, a k-factor approximation for P is a polynomial time algorithm A such that,

$$OPT(I) \leq kA(I)$$

for all instances I of P and $k \geq 1$. The approximation algorithm A is said to have a *performance guarantee* of k. Performance guarantee is also referred to as the *performance ratio*. For minimization problems $A(I) \leq k\ OPT(I)$. Several k-factor approximation algorithms exist for the minimum knapsack problem.

Polynomial time approximation scheme

For a minimization problem, a polynomial time approximation scheme (PTAS) is a family of algorithms A_ε, such that the solution produced by A_ε satisfies $A_\varepsilon(I) \leq (1+\varepsilon)OPT(I)$ for all $\varepsilon > 0$ where I is an instance of the problem at hand and the running time for A_ε is bounded by a polynomial in the size of instance I for a fixed $\varepsilon > 0$.

A_ε is said to be a fully polynomial time approximation scheme (FPTAS) if in addition to the performance guarantee the running time of A_ε is polynomial on the size of the instance I and $1/\varepsilon$. PTAS and FPTAS for maximization problems can be defined analogously. Note that $O(n^{\frac{1}{\varepsilon}})$ is polynomial in n for fixed ε, but is not polynomial in n and $\frac{1}{\varepsilon}$, whereas $O(\frac{n^2}{\varepsilon^3})$ is polynomial in n and $\frac{1}{\varepsilon}$.

1.5 Organization of the thesis

In Chapter 2 we discuss some of the previous works on maximum and minimum knapsack problems. We also examine how the computational efficiency decreases for the multidimensional knapsack as compared to the single dimensional version.

In Chapter 3 we present the MITACS (Mathematics of Information Technology and Complex Systems) internship research work which essentially lead to this thesis. This chapter also portrays how real world problems can be modelled using the knapsack problem.

In Chapter 4 we propose a scaling based FPTAS for the minimum knapsack problem and analyze the complexity of the proposed algorithm. We also propose an ε-approximation scheme for the multidimensional minimum knapsack problem. This algorithm can take exponential time in the worst case. We compare the computational results of the algorithms to other existing approximation algorithms in Chapter 5.

We conclude the thesis in Chapter 6 by outlining some future research possibilities.

1.6 Contributions

The main contribution of this thesis is a scaling based fully polynomial time approximation scheme for the minimum knapsack problem and an ε-approximation scheme for the multi-dimensional minimum knapsack problem and the experimental study of these algorithms.

We compare the performance of these two algorithms with other algorithms for different types of instances to show that these algorithms do well in practice. Theoretically the performance ratio is $\leq 3/2$ for $\varepsilon = 0.5$, but in practice both the algorithms have performance ratio very close to 1 for different types of instances.

Chapter 2

Related Work

In this chapter we present some previous results on the knapsack problem. We describe an existing pseudo-polynomial time algorithm for the maximum knapsack problem and an FPTAS based on scaling and dynamic programming due to Ibarra and Kim [19]. We also describe the approximation factor analysis for the FPTAS.

We describe two approximation algorithms for min-knapsack problem, one greedy [7] and one based on primal-dual schema [4]. The greedy algorithm can be modified to obtain an FPTAS for the minimum knapsack problem.

2.1 Previous work

The concept of dynamic programming can be applied whenever the optimal solution consists of a combination of optimal solutions to the subproblems [2]. This property is known as the principle of optimality [2]. Maximum knapsack problem exhibits the principle of optimality.

Several approximation schemes exist for the $0-1$ maximum knapsack problem. A $2/3$-approximation algorithm with $O(n \log n)$ running time is due to [20]. Several FPTASs exist for the max-knapsack problem [19, 26, 33].

Quite often in a fully polynomial time approximation scheme the time complexity is reduced with a considerable increase in the space requirement. Martello *et al.* [34] pointed out the impracticality of an FPTAS for the knapsack problem due to this increase in space requirement. Afterwards, Kellerer *et al.* [22, 23] proposed FPTAS which reduced space complexity of $O(n+1/\varepsilon^2)$ and described some practical applications of an FPTAS for the knapsack problem.

A natural greedy approach first studied by Gens and Levner [14] for the maximum knapsack problem is to pack the items in decreasing order of the densities (profit per unit weight). The approach proceeds by selecting the items with highest profit per unit weight at each step without exceeding the knapsack capacity. The first item to overflow the knapsack capacity after adding it to the knapsack is referred to as the *split item*. Finding a split item is essential in many of the algorithms for the knapsack problem. If the items are already sorted then this can be done in linear time. But for very large item sets it is an interesting task to find out the split item in linear time without sorting the items. Linear time median find algorithm by Blum *et al.* [3] or the improved one due to Dor *et al.* [9] may be adapted to accomplish this task. Balas *et al.* [1] was the first to use a linear median algorithm for finding a split item in linear time.

For a brief survey of the literature on multidimensional maximum knapsack problem see Chu *et al.* [6]. A section of the survey in [27] also discusses the multidimensional (d-dimensional) maximum knapsack problem. A recent paper by Fréville [12] surveys the main results and focuses on the theoretical properties as well as approximate and exact solutions to multidimensional 0-1 maximum knapsack problem. For some early attempts to solve the d-dimensional knapsack using dynamic programming see [18, 40, 35].

Earlier studies of the d-dimensional maximum knapsack problem were mostly motivated by a budget planning scenario as mentioned in [30], [39] and [40]. The application of

d-dimensional knapsack is diverse in computer science such as scheduling of computer programs [38], allocation of processors and databases in a distributed system [17], allocating shelf-space for consumer products [41].

Dyckhoff [10] developed typology for packing problems where a feasible packing has to satisfy specific geometric constraints of placing rectangular objects in a rectangle of length c_1 and width c_2 where w_{1j} and w_{2j} indicates the length and width of object j. Some recent surveys of the packing problem is given in [28] and [29].

For d-dimensional knapsack the most natural greedy choice is to consider n items given a fixed ordering one after another and put an item in the knapsack if it does not exceed the capacity in any one of the dimensions. Keeping analogy with linear programming technique this is also called as the *primal greedy heuristic* as a feasible solution is maintained throughout the process. Similarly, the *dual greedy heuristic* starts with an infeasible solution by putting all the items in the knapsack and gradually one item after another is removed from the knapsack until the corresponding solutions is feasible. Let the final item t that produces a feasible solution after removing it from the knapsack be termed as *threshold item(t)*. Then a primal greedy heuristic is performed on the items from the first item(1) removed until the threshold item($t-1$) to reduce the gap that might be produced after packing the threshold item t in the knapsack. Fox *et al.* [11] gives a general idea about these techniques. Each item is assigned an efficiency value and then they are sorted in decreasing order for the primal greedy heuristic and in increasing order for the dual greedy heuristic. The efficiency values of the items are calculated differently, see [8], [37] and [31].

2.2 Pseudo-polynomial time algorithm for maximum knapsack

An algorithm is called a pseudo-polynomial time algorithm if the running time of the algorithm is bounded by a polynomial not only in input size n but also in one (or more) of the

other input values, such as profit values or weight values. These types of algorithms may have long running time even for small inputs with large coefficients such as large profit values of items.

One such pseudo-polynomial time algorithm for max-knapsack problem is due to Ibarra and Kim [19]. Given a knapsack instance with n items $1,...,n$ and a knapsack capacity of C. For each $i \in \{1,...,n\}$, w_i and p_i represents the weight and profit of item i respectively. We assume that the profits are integers. Let P be the profit of the most profitable item, i.e., $P = \max_{i \in n} p_i$. The maximum profit that can be achieved by any solution for this instance has an upper bound of nP. Let $S_{i,p}$ denote a subset of items from $\{1,...,i\}$ such that the total profit of the items in the subset is exactly p and total weight is minimum. Let $A(i,p)$ denote the size of the subset $S_{i,p}$ and $A(i,p) = \infty$ if no such set $S_{i,p}$ exists. Size of a subset $S_{i,p}$ is the sum of the weights of the items in the subset $S_{i,p}$.

The following recurrence can be used to compute $A(i,p)$ (hence $S_{i,p}$) for all $i \in \{1,...,n\}$ and $p \in [1,...,nP]$

$$A(i,p) = \begin{cases} A(i-1,p) & \text{if } p_i > p \\ min\{A(i-1,p), w_i + A(i-1,p-p_i)\} & \text{Otherwise.} \end{cases}$$

The optimal solution can be found by choosing the entry with maximum profit and weight not exceeding the knapsack capacity C. This is a pseudo-polynomial time algorithm for maximum knapsack problem as the running time of the algorithm is $O(n^2P)$ which is not polynomial in n. The procedure is explained in Algorithm 2.1:

2.3 FPTAS for max-knapsack

The previously described pseudo-polynomial time algorithm can be modified slightly to obtain an FPTAS for max-knapsack using the scaling technique [19]. The running time of

Algorithm 2.1 Pseudo-polynomial time algorithm for Maximum Knapsack

1. {Let $S_{i,p}$ be the subset of items such that the total profit of the items from $\{1,...,i\}$ is exactly p and $A(i,p)$ denote the size of the subset $S_{i,p}$}.
2. Let $P = max_{i \in n} p_i$.
3. **for** i = 1 to n **do**
4. **for** p=1 to nP **do**
5. **if** $p_i > p$ **then**
6. $A(i,p) = A(i-1,p)$
7. **else**
8. $A(i,p) = min\{A(i-1,p), w_i + A(i-1,p-p_i)\}$ and update $S_{i,p}$
9. **end if**
10. **end for**
11. **end for**
12. Choose the entry with maximum profit and the size not exceeding the knapsack capacity C as the final solution.

the dynamic programming algorithm is bounded by a polynomial not only in n but also the largest profit value P given in the instance. We want to modify the profits of the instance in such a way that the running time of the dynamic program is bounded by polynomial in n and $1/\varepsilon$, where ε is the approximation ratio. The resulting algorithm enables us to find a solution whose profit is at least $(1-\varepsilon)$ times the optimal profit value in time bounded by a polynomial in n and $1/\varepsilon$ for a fixed ε.

For any $\varepsilon > 0$, let $K = \frac{\varepsilon P}{n}$ and for each item i we set $p'_i = \lfloor \frac{p_i}{K} \rfloor$. Using these scaled profits as the profits of the items we apply the dynamic programming to find out the most profitable set S. This set S represents the approximate solution to the problem. The procedure is explained in Algorithm 2.2.

Algorithm 2.2 FPTAS for Maximum Knapsack

1. For a given $\varepsilon > 0$ let $K = \frac{\varepsilon P}{n}$.
2. For each item i, define scaled profit $p'_i = \lfloor \frac{p_i}{K} \rfloor$.
3. Using these scaled profits of items using the dynamic programming of algorithm 2.1 find the most profitable set S without exceeding the knapsack capacity C.
4. Output S as the final solution.

Now we will derive the performance ratio of this algorithm and prove that the algorithm is a fully polynomial time approximation scheme for max-knapsack.

2.3.1 Proof of the performance ratio

Theorem 1 *[Ibarra and Kim, 1975] If S is the solution returned by the Algorithm 2.2 and O' is the optimal solution then, $p(S) \geq p(O')(1-\varepsilon)$, where $p(V)$ is the total profit of items in set V and the running time of the algorithm is $O(\frac{n^3}{\varepsilon})$.*

Proof Let us assume that the optimal solution set for a given knapsack instance is O'. For any item i after rounding down the original profit $p(i)$ can be greater than $Kp'(i)$ by at most K. Let $p(O') = \sum_{i \in O'} p_i$. As the optimal solution can contain at most n items,

$$p(O') \leq Kp'(O') + nK \tag{2.1}$$

In 2.1 $p'(O')$ is the sum of the scaled profits of the items in the set of items O'. As the dynamic programming always returns the best solution,

$$\begin{aligned} p'(S) &\geq p'(O') \\ Kp'(S) &\geq Kp'(O') \end{aligned} \tag{2.2}$$

We also know that,

$$p(S) \geq Kp'(S) \tag{2.3}$$

From 2.2 and 2.3 it follow that,

$$p(S) \geq Kp'(O') \tag{2.4}$$

$$\geq p(O') - nK \qquad \text{by 2.1} \tag{2.5}$$

$$= OPT - \varepsilon P \qquad \text{by definition} \tag{2.6}$$

$$\geq (1-\varepsilon).OPT \tag{2.7}$$

We derive 2.7 from 2.6 by noting that $OPT \geq P$. This follows from the assumption that no item can have a weight value greater than the knapsack size. So, the item having the largest profit (P) value must have weight smaller than the knapsack size. If P was greater than OPT then the item corresponding to this profit could be selected as the only item in the optimal solution to maximize the profit and this would give a feasible solution as the weight value of the item is smaller than the demand.

For each item i the second loop runs from 1 to $n\lfloor\frac{P}{K}\rfloor$. So the running time of the above mentioned algorithm is $O(n^2\lfloor\frac{P}{K}\rfloor)$ or $O(\frac{n^3}{\varepsilon})$ and this is polynomial in both n and $1/\varepsilon$. As the solution is within $(1-\varepsilon)$ factor of the optimal solution, the algorithm is a fully polynomial time approximation scheme for maximum knapsack.

□

2.4 A greedy approximation algorithm and FPTAS for min-knapsack

The original 0-1 knapsack problem (max-knapsack) is widely studied in literature. Several greedy heuristics and ε-approximation schemes have been proposed for this problem [13, 19, 26, 36]. A greedy heuristic for the 0-1 min-knapsack based on the heuristic for max-knapsack by Gens and Levner [14] is described in [7]. This heuristic can be adapted to

obtain an FPTAS for the minimum knapsack problem.

The items are first sorted in nondecreasing order of their relative costs. Relative cost of an item i is defined as the ratio of the cost c_i of the item to the size s_i of the item. So after sorting,

$$c_1/s_1 \leq c_2/s_2 \leq c_3/s_3 \leq ... \leq c_n/s_n$$

The algorithm scans through the items sorted according to their relative cost for the first index k_1 for which,

$$\sum_{i=1}^{k_1} s_i < D \leq \sum_{i=1}^{k_1+1} s_i,$$

where D is the demand. After finding k_1 we have a candidate solution, the sublist $(1,2,...,k_1+1)$ represents the items in the candidate solution. We call the set of items for which the total weight of the set is less than the demand as the set of *small items* then we can write, $S_1 = (1,2,...,k_1)$ as the first set of *small items*. The candidate solution can be written as, $S_1 \cup \{k_1+1\}$.

Step 1: Now starting from k_1+2 we scan through the items and let k_2 be the next item for which $\sum_{i=1}^{k_1} s_i + s_{k_2} < D$. So for all the items $j \in [k_1+2,...,k_2-1]$ the following holds,

$$\sum_{i=1}^{k_1} s_i + s_j \geq D$$

We denote all the items k_1+1 to k_2-1 as the *big items* and let the first set of *big items*

be $B_1 = (k_1 + 1, ..., k_2 - 1)$. Now all sets $S_1 \cup \{j\}, j \in [k_1 + 2, ..., k_2 - 1]$, are candidate solutions.

Step 2: Now find the first index $k_3 \geq k_2$ for which the following holds,

$$\sum_{i=1}^{k_1} s_i + \sum_{i=k_2}^{k_3} s_i < D \leq \sum_{i=1}^{k_1} s_i + \sum_{i=k_2}^{k_3+1} s_i$$

Now we can define a second set of *small items* $S_2 = \{k_2, ..., k_3\}$. So, $S_1 \cup S_2 \cup \{k_3 + 1\}$ is also a candidate solution. Now we repeat steps 1 and 2 until the end of the list using k_{2i+1} instead of k_1 and k_{2i+2} instead of k_2 in the i^{th} iteration. The solution will be the one with smallest cost amongst all the candidate solutions. The sorting step takes $O(n \log n)$ and the later steps take $O(n)$ time. The algorithm mentioned above has an approximation ratio of 2 [7]. The procedure is shown in Algorithm 2.3:

Algorithm 2.3 Greedy approximation algorithm for Minimum Knapsack

1. Sort n items in list L in nondecreasing order of their relative costs.
2. Find the first index k_1 which
   $\sum_{i=1}^{k_1} s_i < D \leq \sum_{i=1}^{k_1+1} s_i$.
   Denote $S_1 = (1, 2, ..., k_1)$ as the first set of small items and sublist $S_1 \cup \{k_1 + 1\}$ as a candidate solution.
3. **repeat**
4. Find all items before k_2 such that, for all the items $j \in [k_1 + 2, ..., k_2 - 1]$ these two inequalities hold $\sum_{i=1}^{k_1} s_i + s_{k_2} < D$ and $\sum_{i=1}^{k_1} s_i + s_j \geq D$.
   Denote $B_1 = (k_1 + 1, ..., k_2 - 1)$ as the first set of big items and all sets $S_1 \cup \{j\}$, $j \in [k_1 + 2, ..., k_2 - 1]$ as candidate solutions.
5. Find the first item $k_3 \geq k_2$ such that,
   $\sum_{i=1}^{k_1} s_i + \sum_{i=k_2}^{k_3} s_i < D \leq \sum_{i=1}^{k_1} s_i + \sum_{i=k_2}^{k_3+1} s_i$.
   Denote $S_2 = \{k_2, ..., k_3\}$ as the second set of small items and $S_1 \cup S_2 \cup \{k_3 + 1\}$ as a candidate solution.
6. **until** The end of list L using k_{2i+1} instead of k_1 and k_{2i+2} instead of k_2 in the i^{th} iteration

The algorithm described above can be refined to improve the approximation ratio to $3/2$ in

$O(n^2)$ time [7].

2.5 A primal dual approximation algorithm for min-knapsack

The first primal-dual schema based approximation algorithm for the one dimensional $0-1$ min-knapsack problem was given in [4]. The paper uses the flow cover inequalities developed in [5] to strengthen the integrality gap of linear programming relaxation of the natural integer programming formulation for the minimum knapsack problem.

Consider a set of items F where each item $i \in F$ has a cost c_i and a size s_i. The goal is to select a minimum cost subset of items, $A \subseteq F$, such that the weight of A, s_A meets the specified demand D, i.e. at least as big as the specified demand. A natural IP formulation is,

$$\begin{aligned} \text{IP:} \qquad & \min \sum_{i \in F} c_i y_i & (2.8) \\ & \text{subject to,} \sum_{i \in F} s_i y_i \geq D \\ & y_i \in \{0,1\} \qquad\qquad \forall i \in F \end{aligned}$$

Here y_i are binary variables that indicate whether and item is chosen in the solution or not. The corresponding linear programming relaxation for the integer program is,

$$\text{LP:} \qquad \min \sum_{i \in F} c_i y_i$$

$$\text{subject to,} \sum_{i \in F} s_i y_i \geq D$$

$$y_i \geq 0 \qquad \forall i \in F$$

Integrality gap for a minimization problem is defined as the ratio of the optimal solution to the integer program to the optimal solution to the linear programming relaxation. The integrality gap between the IP and the LP relaxation above can be as bad as D. If we consider 2 items where $s_1 = D-1$, $c_1 = 0$, $s_2 = D$ and $c_2 = 1$. The only feasible integer solution chooses both items and incurs a cost of 1, whereas the LP solution can set $y_1 = 1$ and $y_2 = 1/D$ and incurs a cost of $1/D$. In this case the integrality gap becomes as bad as D. This situation can be remedied using flow cover inequalities introduced in [5].

Consider a subset of items $A \subseteq F$ such that $s(A) < D$, and let $D(A) = D - s(A)$, where $s(A) = \sum_{i \in A} s_i$. If all the items in set A are selected then we must select enough items in $F \backslash A$ to meet the residual demand $D(A)$. This is essentially another min-knapsack problem where the items are restricted to only $F \backslash A$ and demand is now $D(A)$. Also the size of every item in $F \backslash A$ can be restricted to be no greater than the demand without changing the set of feasible integer solutions, so we set $s_i(A) = min\{s_i, D(A)\}$ for every item $i \in F \backslash A$. Now define the following linear programming formulation called Minimum Knapsack Primal (MKP),

$$\text{MKP:} \qquad \min \sum_{i \in F} c_i y_i$$

$$\text{subject to,} \sum_{i \in F \setminus A} s_i(A) y_i \geq D(A) \qquad \forall A \subseteq F$$

$$y_i \geq 0 \qquad \forall i \in F$$

The dual of this *LP* is called Minimum Knapsack Dual (MKD),

$$\text{MKD:} \qquad \max \sum_{A \subseteq F} D(A) v(A)$$

$$\text{subject to,} \sum_{A \subseteq F : i \notin A} s_i(A) v(A) \leq c_i \qquad \forall i \in F$$

$$v(A) \geq 0 \qquad \forall A \subseteq F$$

In this dual problem $v(A)$ is the dual variable corresponding to the constraints in the primal problem MKP and the constraints in the dual correspond to the primal variable y_i.

The primal dual algorithm described in [4] is as follows,

Algorithm 2.4 Primal-Dual for Minimum Knapsack

```
y, v ⇐ 0
A ⇐ Ø
while D(A) > 0 do
    Increase v(A) until a dual constraint in MKD becomes tight for an item i
    y_i ⇐ 1
    A ⇐ A ∪ {i}
end while
S ⇐ A
```

Theorem 2 *[Carnes and Shmoys, 2008]Algorithm 2.4 terminates with a solution of cost no greater than* $2.OPT_{IP}$.

The above mentioned relaxation can have an integrality gap of 2 which is better than the straight forward relaxation where the integrality gap can be as bad as D. Therefore the bound of 2 is tight. It should be noted that MKD contains exponentially many variables however Algorithm 2.4 runs in polynomial time.

2.6 Generalization of Gens and Levner heuristic for d-dimensional min knapsack

We describe a generalization of an algorithm of Gens and Levner [14] for the d dimensional min knapsack problem. This algorithm was communicated to us by [16]. Without loss of generality, we assume that the right hand side of each capacity constraint is 1.

Let $w_j(i)$ denote the weight of item i in dimension j and $w(i) = \sum_{j=1}^{d} w_j(i)$. Let S denote any set of elements, then $w_j(S)$ denotes the sum of the attributes of the items is set S in dimension j. The relative cost of an item i is given by $r_i = \frac{c_i}{w(i)}$.Let R_i be the subset of items $\{1,...,i-1\}$. The running time of Algorithm 2.5 is in $O(n^2)$.

Algorithm 2.5 Greedy algorithm for d dimensional min knapsack

1. Order The elements in the non-decreasing order of their relative cost.
2. For each i from 1 to n color the element as follows:
   $w_d(R_i) + w_d(i) < 1$ for some dimension $d \Rightarrow color(i) = R$
   $w_d(R_i) + w_d(i) \geq 1$ for all dimensions $d \Rightarrow color(i) = B$
3. For each item j colored B consider the solution $R_i \cup j$, where R_i is the smallest index R colored set such that $w_d(R_i) + w_d(j) \geq 1$ for all dimensions d. Pick the cheapest such solution.

Chapter 3

MITACS Internship program

At the beginning of the second year of my M. Sc. program in May, 2008 I was selected for the MITACS (Mathematics of Information Technology and Complex Systems) internship program (www.mitacs.ca). This internship program eventually lead to my M. Sc. thesis. This chapter is based on the report submitted to MITACS upon completion of the project.

3.1 Goal of the internship program

Canadian Pacific Logistics Solutions (CPLS-www.cpls.ca) was a partner organization in this project and they are contractually required to analyze the fleet of railcars used by DIAGEO (www.diageo.com) for their bulk whisky business. DIAGEO is a client of CPLS. DIAGEO leases railcars from Canadian Pacific Railway to transport its products from one location to the other. The goal of the internship project was to analyze the data for the transportation of the products of DIAGEO and suggest any improvement in the fleet organization in such a way that the total number of rail cars used is minimized and minimize the cost thereby.

3.2 Problem specification and other details

As we mentioned earlier, the goal of the project was to analyze fleet of DIAGEO and reduce the number of railcars required for transportation of bulk liquid. A Reduction in number of railcars would reduce the cost of leasing the railcars and thereby increase the overall revenue. Contractually CPLS will be paid 10% of any of the savings identified by DIAGEO.

The initial step was to collect relevant data to generate statistical information about the fleet used by DIAGEO. We collected data from a database for the time period of May 1^{st} 2007 to May 31^{st} 2008. The data included the product specifications, loaded dwell time at the origin, loaded transit time, destination dwell time, empty transit time, recommended cycle time and optimal cycle time. This information was used to calculate a lower bound on the number of railcars needed to transfer the bulk liquid. Product specifications included different types of products such as whisky, grain neutral spirit (GNS) etc. The loaded dwell time at the origin indicates how long a railcar has to wait at the origin to load the products , loaded transit time indicates how long it takes for a railcar to go from origin to the destination carrying the product, destination dwell time indicates how long a railcar has to wait to unload the products at the destination, empty transit time indicates how long it takes for a railcar to go back to the origin after unloading at the destination, recommended cycle time allows for 6 days of dwell at each end, optimal cycle time allows for 3 days of dwell at each end.

The actual demand for a source-destination pair (lane) was not explicitly specified in the work-sheet *Detail*, but we calculated it (for each lane) by adding up all the volume of the products transported in that lane by different railcars. The capacity of the railcars were obtained from the *Cost Per Car* work-sheet. We found the railcars assigned to a specific

lane from the *Fleet List* work-sheet. After collecting data from different work-sheets we had to perform some data cleaning tasks, like converting the unit of all the volumes to Liter from Gallon and Barrel. There were several stages of data cleaning and manipulating. This refined data was used in our calculations.

We modelled the problem as an integer program with the objective to compute a lower bound on the number of cars with several constraints. Later it was discovered that the problem is decomposable and it can be solved efficiently by dynamic programming. The integer program follows:

$$\text{IP2: minimize} \sum_{p \in P} \sum_{k \in T} x_{pk}$$

$$\text{subject to} \sum_{k \in T} C_k x_{pk} \geq d_p \qquad \forall p \in P$$

$$W_p \geq A_{pk} y_{pk} \qquad \forall p \in P \text{ and } \forall k \in T$$

$$x_{pk} \geq 0 \text{ and integer}$$

$$y_{pk} = \begin{cases} 1 & \text{if vehicle type } k \text{ is used in lane } p \\ 0 & \text{Otherwise} \end{cases}$$

Here T is the set of all types of railcars and P is the set of all origin-destination pairs. C_k is the capacity of a railcar of type k and variable x_{pk} indicates how many cars of type k are being used in origin-destination pair p. d_p is the demand for the origin-destination pair p. W_p is the maximum weight that can be carried in lane p and A_{pk} is the actual weight carried by a car of type k in lane p.

In general, integer programs can not be solved efficiently. However, given the objective function the problem at hand can be solved efficiently using a dynamic programming ap-

proach applicable to knapsack instances. The data at hand was used to generate three different variants of the minimum knapsack problem and results were computed for the different variants. Another important thing to notice in the work-sheets was that, although one car was assigned to only one lane as in the *FleetList* worksheet, it was actually being used in several lanes as we see in the *Detail* worksheet. Our focus was not on scheduling the cars but on packing the cars as efficiently as possible. Therefore, our computation gives a lower bound on the number of railcars needed.

3.3 Relation with the knapsack problem

The integer program IP2 can be reduced to the first variant of the knapsack problem, also known as the 0-1(binary) knapsack problem. Each car is repeated enough number of times based on either the cycle times or the demands to create a $0-1$ knapsack instance. Once the $0-1$ knapsack instance is created, the optimal solution is found by a dynamic programming algorithm [19]. This algorithm is a pseudo-polynomial time algorithm as it depends not only on number of items but also on the value of the profits of the items. Since all the profits are unity this algorithm runs in polynomial time on our inputs.

3.4 Generation of different knapsack instances

We generated three different types of knapsack instances. Next we describe the generation of three different types of knapsack instances. Specifications of the machine on which we ran all the experiments are,

Manufacturer: Acer.

Model: Aspire 5920G.

Processor:Intel (R) core (TM) Duo CPU T5550 at 1.83 GHz.

Memory (RAM): 2.0 GB

3.4.1 Type based calculation

In this approach we assume that, the cars can be classified into five different types. The types are determined from the data provided in the worksheet *CostPerCar*. Although cars of the same type have different capacities, we assume that all the cars of the same type have the same capacity. The capacity that we have assumed for each type of car is the minimum capacity of all the cars of that type. It is also possible to assume average capacity as the capacity of the car and this would lead to a reduction in the number of shipments.

For each origin and destination pair the cars are repeated enough number of times to be able to meet the demand for the lane. For example, if the demand from location a to location b is 100 units and the capacity of a railcar of type 1 is 5 units and type 2 is 10 units then railcar of type 1 is repeated 20 times and railcar of type 2 is repeated 10 times to carry out the whole demand in between location a and b. The knapsack instance generated in this way is solved efficiently using the dynamic programming algorithm. For each lane we generate a knapsack instance that is solved independently of other instances using the dynamic programming algorithm. In this approach the knapsack with minimum number of items was for the lane Lawrenceberg to Menlo-Park and the number of items in the knapsack is 22, and the solution matrix has dimensions 105 by 105 for this lane. The knapsack with maximum number of items was for the lane Gimli to Amherstburg and the number of items in the knapsack is 225. In this case the solution matrix has dimension 1182 by 1182. The total running time for this approach is 9.97499891746 seconds. This time also includes some file I/O (Input/Output) operations.

3.4.2 Cycle time based calculation

In the second approach, each car is treated separately. We take into account the capacity of each of the cars unlike in the previous one in which we considered only the capacity of

a particular type only but not the different capacities of the cars of the same type. Cycle time for a lane is defined as the total time required for a car to start from a location, reach its destination and return to the origin, thus completing a cycle. The worksheet *FleetList* had three different cycle times. Optimal cycle time allowed 3 days of dwell at each end and the recommended cycle time allowed 6 days of dwell at each end. Actual cycle time is the actual time taken by the cars to complete the cycle on a specific lane. The cars that are being used on a lane are repeated enough number of times, to complete maximum cycles in the total time period ($\lfloor$396 days/cycletime$\rfloor$). The same calculation is performed on each lane for the optimal cycle-time and recommended cycle-time. The solution matrix with maximum dimensions was for the lane Gimli-Amherstburg and the dimensions were 900 by 900. Total number of items in this knapsack was 226. The knapsack instance with minimum number of items was for lane Lawrenceberg to Menlo-Park and the solution matrix dimensions were 40 by 40. The total number of items in the knapsack for this lane was 20. The total running time of this approach is 12.7509524954 seconds. This time includes some file I/O operation as well.

3.4.3 Original data based calculation

This approach is an extremely restrictive one. Here we assume that a car can be repeated at most the number of times it is being used currently (as specified in the database) in the lane. It is different from the previous two approaches in the sense that it can not use a car more than the number it was used in the original data. Whereas in the previous two approaches there was no such restrictions on the number of times a car can be used. Gimli-Amherstburg lane has the solution matrix with maximum dimensions of 251 by 251. The number of items in the knapsack for this lane is 228. The minimum solution matrix was of dimensions 21 by 21 for the lane Lawrenceberg to Menlo-Park and the number of items in the corresponding knapsack is 20. The total running time of this approach is

8.46195144914 seconds including some file I/O operations.

3.5 Performance comparison of different instances

3.5.1 Type based calculation

In the original work-sheet that was provided by CPLS the actual number of cars, optimal number of cars and the recommended number of cars were calculated by dividing the total shipments on a lane by maximum number of times a car can be repeated based on the average cycle-time, optimal cycle-time and recommended cycle-time respectively. The final figure is obtained by dropping the fractional part.

In our calculations we took the ceiling of the cycle-time and then divided the total time (396 days) by this number and took the floor. We have taken floor because if a car cannot complete a cycle then we do not take it into account. Finally, we divide the total number of shipments on that lane by the number computed as above and took the ceiling to obtain the number of cars required. So the figures in the original worksheet and our worksheet may look a little bit different. For details on the types of cars and the capacities please refer to the previous section.

In the first approach, after obtaining an optimal solution to IP2 (using the dynamic programming approach) we get the total number of shipments for all the pairs of origin and destination. We then calculate the number of cars required for each lane in the time duration May 1^{st} 2007 to May 31^{st} 2008 based on the cycle time. The same calculation is done for the optimal cycle time and for the recommended cycle time of the lane. Figure 3.1 shows the spreadsheet with the old values and the new values. From the spreadsheet we can see that, the old number of shipments was 1124 and the new (calculated) number of shipments is 1080. This shows that, the current technique used to assign cars to lanes is

LANE	Actual # of shipments(old)	Average cycle time	Actual # Cars(old)	Optimal Cycle time	Optimal# cars(old)	Recommended Cycle time	Recommended # of cars(old)	Actual # of shipments(new)	Actual (new)	Optimal # of Cars(new)	Recommended # of Cars(new)
GIMLI - AMHERSTBURG	251	52.68	36	28.58	20	34.58	23	225	33	18	21
GIMLI - RELAY	71	54.55	11	39.09	8	45.09	9	67	10	8	9
GIMLI - VALLEYFIELD	11	59.08	2	34.16	1	40.16	2	11	2	1	2
GIMLI - HIRAM WALKER	107	59.06	18	31.21	9	37.21	11	98	17	9	10
GIMLI TOTAL	**440**		**67**		**38**		**45**	**401**	**62**	**36**	**42**
HIRAM WALKER - AMHERSTE	26	42	3	18	2	24	2	25	3	2	2
HIRAM WALKER TOTAL	**26**		**3**		**2**		**2**	**25**	**3**	**2**	**2**
LA CROSSE - NMB - CH/	216	33.51	20	23.65	14	29.65	17	196	18	13	16
LA CROSSE - NMB - PLA	177	44.93	23	23.36	12	29.36	14	171	22	11	14
LA CROSSE TOTAL	**393**		**43**		**26**		**31**	**367**	**40**	**24**	**30**
LAWRENCEBURG - AMHERST	71	53.32	11	29.76	6	35.76	8	78	12	6	8
LAWRENCEBURG - MENLO PA	21	63.72	4	44.93	3	50.93	3	22	4	3	4
LAWRENCEBURG - RELAY	26	57.37	5	27.48	2	33.48	3	26	5	2	3
LAWRENCEBURG TOTAL	**118**		**20**		**11**		**14**	**126**	**21**	**11**	**15**
LETHBRIDGE - GIMLI	41	42.49	5	29.7	4	35.7	5	40	5	4	4
LETHBRIDGE - HIRAM WALK	25	64.18	5	36.74	3	42.74	3	28	5	3	4
LETHBRIDGE TOTAL	**66**		**10**		**7**		**8**	**68**	**10**	**7**	**8**
RELAY - MENLO PARK	47	56.57	8	40.47	6	46.47	6	47	8	6	6
RELAY TOTAL	**47**		**8**		**6**		**6**	**47**	**8**	**6**	**6**
WASHINGTON - PLAINFIELD	34	47.6	5	35.45	4	41.45	4	46	6	5	6
WASHINGTON TOTAL	**34**		**5**		**4**		**4**	**46**	**6**	**5**	**6**
TOTAL CARS	**1124**		**156**		**94**		**110**	**1080**	**150**	**91**	**109**

Figure 3.1: Spreadsheet for cars repeated based on the type

quite good, and there is not much room for improvement. Note that the optimal solution of the IP2 is a lower bound on the number of cars used. The improvement is 3.91% in the number of shipments. For the actual number of cars the old number is 156, and the new number is 150. Here the improvement is 4.0% in terms of the actual number of cars. Recommended number of cars in the old calculation is 110 and recommended number of cars in our calculation is 109. Optimal number of cars in the old calculation is 94 and in our calculation the optimal number of cars is 91.

3.5.2 Cycle-time based calculation

In this approach we take into account only those cars that are currently being used for transportation in the lane, as opposed to the assumption in the previous section that each type of car is available on every lane. This method restricts our computation to only a subset of the total cars and then finds out the solution of the knapsack instance formed in this way with enough number of items. In Figure 3.2, we see that the actual number of shipments (obtained by the calculation) is 1028 whereas the corresponding number in the worksheet provided is 1124. In this case the improvement is 8%. The actual number of cars is 132 as opposed to 156. The improvement here is 15.38%. As the calculation is based on cycle-time(s) the number of shipments also changes when we calculate it for recommended cycle time. The total number of shipments for the recommended cycle time is 1039 and the total number of cars in this case is 106. Improvement is 3.6% in case of recommended number of cars. The total number of shipments for the optimal cycle time is 1009 and the total number of cars in this case is 89. For the optimal number of cars the improvement is 5.31%.

LANE	Actual # of shipments (old)	Average cycle time	Actual # Cars (old)	Optimal Cycle time	Optimal# cars (old)	Recommended Cycle time	Recommended # of cars (old)	Actual # of shipments (new)	Actual (new)	Optimal # of shipments	Optimal # of Cars (new)	Recommended # of shipments	Recommended # of Cars(new)
GIMLI - AMHERSTBURG	251	52.68	36	28.58	20	34.58	23	214	27	205	16	226	20
GIMLI - RELAY	71	54.55	11	39.09	8	45.09	9	68	10	71	8	69	9
GIMLI - VALLEYFIELD	11	59.08	2	34.16	1	40.16	2	9	2	9	2	9	2
GIMLI - HIRAM WALKER	107	59.06	18	31.21	9	37.21	11	85	13	89	8	84	8
GIMLI TOTAL	**440**		**67**		**38**		**45**	**376**	**52**	**374**	**34**	**388**	**39**
HIRAM WALKER - AMHERSTBURG	26	42	3	18	2	24	2	20	3	20	2	20	2
HIRAM WALKER TOTAL	**26**		**3**		**2**		**2**	**20**	**3**	**20**	**2**	**20**	**2**
LA CROSSE - NMB - CHA	216	33.51	20	23.65	14	29.65	17	210	18	201	13	210	16
LA CROSSE - NMB - PLA	177	44.93	23	23.36	12	29.36	14	173	22	171	11	172	14
LA CROSSE TOTAL	**393**		**43**		**26**		**31**	**383**	**40**	**372**	**24**	**382**	**30**
LAWRENCEBURG - AMHERSTBURG	71	53.32	11	29.76	6	35.76	8	62	8	63	7	64	8
LAWRENCEBURG - MENLO PARK	21	63.72	4	44.93	3	50.93	3	20	4	20	4	20	4
LAWRENCEBURG - RELAY	26	57.37	5	27.48	2	33.48	3	25	5	23	3	23	3
LAWRENCEBURG TOTAL	**118**		**20**		**11**		**14**	**107**	**17**	**106**	**14**	**107**	**15**
LETHBRIDGE - GIMLI	41	42.49	5	29.7	4	35.7	5	37	5	36	3	37	4
LETHBRIDGE - HIRAM WALKER	25	64.18	5	36.74	3	42.74	3	23	4	23	3	23	4
LETHBRIDGE TOTAL	**66**		**10**		**7**		**8**	**60**	**9**	**59**	**6**	**60**	**8**
RELAY - MENLO PARK	47	56.57	8	40.47	6	46.47	6	44	6	44	5	44	6
RELAY TOTAL	**47**		**8**		**6**		**6**	**44**	**6**	**44**	**5**	**44**	**6**
WASHINGTON - PLAINFIELD	34	47.6	5	35.45	4	41.45	4	38	5	34	4	38	4
WASHINGTON TOTAL	**34**		**5**		**4**		**4**	**38**	**5**	**34**	**4**	**38**	**6**
TOTAL CARS	**1124**		**156**		**94**		**110**	**1028**	**132**	**1009**	**89**	**1039**	**106**

Figure 3.2: Spreadsheet for cars repeated based on the cycle time

LANE	Actual # of shipments(old)	Average cycle time	Actual # Cars(old)	Optimal Cycle time	Optimal# cars(old)	Recommended Cycle time	Recommended # of cars(old)	Actual # of shipments(new)	Actual (new)
GIMLI - AMHERSTBURG	251	52.68	36	28.58	20	34.58	23	228	70
GIMLI - RELAY	71	54.55	11	39.09	8	45.09	9	65	29
GIMLI - VALLEYFIELD	11	59.08	2	34.16	1	40.16	2	10	9
GIMLI - HIRAM WALKER	107	59.06	18	31.21	9	37.21	11	86	43
GIMLI TOTAL	**440**		**67**		**38**		**45**	**389**	**151**
HIRAM WALKER - AMHERSTBURG	26	42	3	18	2	24	2	22	22
HIRAM WALKER TOTAL	**26**		**3**		**2**		**2**	**22**	**22**
LA CROSSE - NMB - CHA	216	33.51	20	23.65	14	29.65	17	211	50
LA CROSSE - NMB - PLA	177	44.93	23	23.36	12	29.36	14	173	49
LA CROSSE TOTAL	**393**		**43**		**26**		**31**	**384**	**99**
LAWRENCEBURG - AMHERSTBURG	71	53.32	11	29.76	6	35.76	8	65	12
LAWRENCEBURG - MENLO PARK	21	63.72	4	44.93	3	50.93	3	20	4
LAWRENCEBURG - RELAY	26	57.37	5	27.48	2	33.48	3	26	9
LAWRENCEBURG TOTAL	**118**		**20**		**11**		**14**	**111**	**25**
LETHBRIDGE - GIMLI	41	42.49	5	29.7	4	35.7	5	39	8
LETHBRIDGE - HIRAM WALKER	25	64.18	5	36.74	3	42.74	3	24	8
LETHBRIDGE TOTAL	**66**		**10**		**7**		**8**	**63**	**16**
RELAY - MENLO PARK	47	56.57	8	40.47	6	46.47	6	44	8
RELAY TOTAL	**47**		**8**		**6**		**6**	**44**	**8**
WASHINGTON - PLAINFIELD	34	47.6	5	35.45	4	41.45	4	34	5
WASHINGTON TOTAL	**34**		**5**		**4**		**4**	**34**	**5**
TOTAL CARS	**1124**		**156**		**94**		**110**	**1047**	**326**

Figure 3.3: Spreadsheet for cars repeated based on the original data

3.5.3 Original data based calculation

In the third approach, the number of shipments is 1047 as opposed to 1124 in the old shipment number (see Figure 3.3). The improvement here is 6.85%. But if we examine in the actual number of cars required then we find that this number 326 is greater than the actual number of cars in the original data 156. This is because many of the cars are used in more than one lanes and being double counted. In fact the actual number of cars used is the same number as being currently used by DIAGEO. The solution to the integer program shows that there is a better way to pack the products so as to minimize the number of shipments. Our focus was on using the knapsack problem to solve the packing problem so it actually packs the cars in the tightest way possible using the cars available on that lane. If we look at the original data provided for the time duration from May 2007 to May 2008 we get an explanation for the double counting. In the Fleet List work-sheet we find that there is a total of 16 cars assigned on the Gimli-Hiramwalker lane, but when we look at the Detail worksheet we can see that there is a total of 44 cars assigned to this lane. It is clear from the data that some cars are being used in more than one lane. This explains the result.

3.6 Analysis

After analyzing the three different approaches and looking at the experimental results we discovered that the second approach is a better way to generate knapsack instances and produces better solutions (the number of shipments and the number of cars). In the second case we restrict the cars to the lanes where they are being used right now but we let it cycle maximum number of times. The current approach used by CPLS and DIAGEO is a good one which is using the cars almost as efficiently as possible (to meet the demand). Albeit there is small room for improvement that we can achieve through this technique. Our

focus was mainly on packing the cars to meet the demand and not scheduling the cars. It was noted in our meetings with CPLS that the problem of scheduling the trains is handled by a different department. The scheduling process can possibly introduce inefficiencies and increase the number of cars needed and further reduce the gap between the number computed and the current fleet size of DIAGEO.

Chapter 4

Minimum knapsack problem

We propose a fully polynomial time approximation scheme for the minimum knapsack problem based on scaling and also a dynamic programming approach to the multidimensional minimum knapsack problem. Note that an FPTAS is implicit in the work of Csirik *et al.* [7] however our approach is different and the analysis is considerably simpler and does well in practice.

4.1 FPTAS for minimum knapsack

Given n items $1..n$ with cost c_i and size s_i each, and a demand D the problem asks to find a subset of items which meets the demand and the cost is minimum possible. In other words, the task is to find a subset S of items such that the total size is at least the demand D and the total cost of all the items is minimum possible.

We propose a fully polynomial time approximation scheme for the minimum knapsack problem which is similar to the FPTAS for the maximum knapsack problem in [19]. This algorithm also uses dynamic programming technique based on costs associated with the items and the scaling technique.

First we describe a pseudo polynomial time dynamic programming algorithm for the minimum knapsack problem. We denote $S(i,c)$ as the set of items from $1..i$ whose total cost

is exactly c and size is the *maximum* possible. Let $A(i,c)$ denote the size of the set $S(i,c)$, $A(i,c)$ can be computed using the following recurrence,

$$A(i,c) = \begin{cases} A(i-1,c) & \text{if } c_i > c \\ max\{A(i-1,c), s_i + A(i-1,c-c_i)\} & \text{Otherwise} \end{cases}$$

Set $S(i,c)$ can be updated according to the recurrence relation mentioned above for $A(i,c)$. We can find the cost of the optimal solution by looking at the entry with minimum cost c such that the sizes of the items in $S(n,c)$ is at least D. If $C = \max\{c_i\}$ then the maximum cost that can be incurred by a solution is nC. So the running time of the dynamic programming algorithm is $O(n^2C)$ as it was in the case of maximum knapsack problem. This is a pseudo polynomial time algorithm for the min knapsack problem.

Now we describe an FPTAS for the minimum knapsack problem. Unlike the FPTAS for the max knapsack problem here we first order the items such that $c_1 \leq c_2 \leq ... \leq c_n$. In this case we need to order the items because unlike the maximum knapsack problem here we can not assume that the largest cost C is smaller than or equal to the optimal solution OPT because this largest cost item may not have been included in the optimal solution. Now for each i in $[1..n]$ we solve the subproblem P_i involving items $[1..i]$ using the dynamic programming algorithm after scaling the costs by $k_i = \varepsilon c_i/i$. After scaling the cost of each item j becomes $c'_j = \lfloor c_j/k_i \rfloor$. Now we have to find out the minimum cost feasible solution over all the subproblems using the above mentioned dynamic program. The FPTAS is described in algorithm 4.1,

4.1.1 Proof of the performance ratio

Theorem 3 *If A is the solution returned by the algorithm 4.1 and O is the optimal solution then, $c(A) \leq c(O)(1+\varepsilon)$, where $c(V)$ is the total cost of items in set V and the running time*

Algorithm 4.1 FPTAS for Minimum Knapsack

1. Order n items such that $c_1 \leq c_2 \leq ... \leq c_n$.
2. **for** i = 1 to n **do**
3. For the subproblem P_i involving items $[1..i]$ and for a fixed $\varepsilon > 0$, let $k_i = \frac{\varepsilon c_i}{i}$.
4. For each item j define cost $c'_j = \lfloor c_j / k_i \rfloor$.
5. With these scaled costs using dynamic programming find the minimum cost feasible solution.
6. **end for**
7. Find the minimum cost feasible solution over all the subproblems and output this as the final solution.

of the algorithm is $O(\frac{n^4}{\varepsilon})$.

Proof Let $c_{i'}$ be the largest cost in the optimal solution O. Now consider the subproblem $P_{i'}$, all the items in this subproblem have cost $\leq c_{i'}$. We will use scaling factor $k = k_{i'}$ in further discussions. Let $A_{i'}$ be the solution returned by the dynamic programming algorithm for the subproblem $P_{i'}$. The total scaled cost of a set S of items is denoted by $c'(S)$ where scaling is based on the largest cost item in the set S. The dynamic programming algorithm returns the cheapest cost solution, therefore, the total scaled costs of items in set $A_{i'}$,

$$c'(A_{i'}) \leq c'(O) \tag{4.1}$$

Here $c'(O)$ is the sum of all the scaled costs of the items in the set O and scaling is performed on the basis of the largest cost in the set. After scaling down by k the following holds for the scaled cost of an item i,

$$kc'_i + k \geq c_i \tag{4.2}$$

Therefore, the cost of all the items in $A_{i'}$,

$$c(A_{i'}) \leq \sum_{j \in A_{i'}} (kc'_j + k) = kc'(A_{i'}) + k|A_{i'}| \tag{4.3}$$

As the solution returned by the dynamic programming algorithm for the subproblem $P_{i'}$ can contain at most i' items, therefore the following holds,

$$k|A_{i'}| \leq ki' = c_{i'}\varepsilon \tag{4.4}$$

Using 4.2 we can write,

$$c(O) \geq kc'(O)$$

Therefore, the performance ratio,

$$\frac{c(A)}{c(O)} \leq \frac{c(A_{i'})}{c(O)} \tag{4.5}$$

The above inequality holds because the cost of the final solution $c(A)$ found by the dynamic programming algorithm can be at most the cost of the solution found at an intermediate stage subproblem $c(A_{i'})$, i.e. , $c(A) \leq c(A_{i'})$. Using 4.5 we can write,

Item number	Cost	Weight
1	3000	7
2	1000	8
3	2000	4
4	5000	5
5	2000	6
6	4000	5

Table 4.1: Simple knapsack instance

$$\frac{c(A)}{c(O)} \leq \frac{kc^{'}(A_{i'}) + c_{i'}\varepsilon}{\max\{kc^{'}(O), c(O)\}} \leq 1+\varepsilon$$

The last inequality follows from 4.1 and the fact that, $c_{i'} \leq c(O)$. Therefore, we can write,

$$c(A) \leq c(O')(1+\varepsilon)$$

For the i^{th} subproblem the maximum profit is c_i and the scaled profit is $c_i^{'} = \lfloor \frac{c_i}{k_i} \rfloor$, where $k_i = \frac{\varepsilon c_i}{i}$. So, for the i^{th} subproblem the running time is $O(i^2 c_i^{'})$, which can be rewritten as $O(i^2 \frac{c_i}{k_i})$ or $O(i^2 \frac{c_i i}{\varepsilon c_i})$. For a total of n subproblems the running time is $O(\frac{1}{\varepsilon} n^4)$.

□

4.1.2 Illustration of the algorithm using an example knapsack instance

Now we illustrate the above mentioned algorithm using a simple knapsack instance. The example knapsack instance consists of 6 items given in Table 4.1.

The algorithm first orders the items in increasing order based on their costs. After ordering the sequence of the items and the corresponding costs and weights (sizes) are given in Table

Item number	Cost	Weight
2	1000	8
3	2000	4
5	2000	6
1	3000	7
6	4000	5
4	5000	5

Table 4.2: Simple knapsack instance after ordering based on costs

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	8	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	0	∞	8	∞	4	∞	12	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	0	∞	8	∞	6	∞	14	∞	10	∞	18	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
4	0	∞	8	∞	6	∞	14	7	10	15	18	13	∞	21	∞	17	∞	25	∞	∞	∞
5	0	∞	8	∞	6	∞	14	7	10	15	18	13	∞	21	∞	19	12	25	20	23	18
6	0	∞	8	∞	6	∞	14	7	10	15	18	13	5	21	13	19	12	25	20	23	18

Table 4.3: Solution matrix for P_6 showing costs from 1-20

4.2 and the demand D is 20. The algorithm will have 6 different subproblems. Subproblem P_i will have items $1..i$ for each i in 1...6.

In the algorithm we let $\varepsilon = 0.5$. Hence, for the subproblem P_1 the value of k_1 is 500 because we computed k_i as $k_i = \varepsilon c_i/i$. For P_1, $i = 1$ and c_1 =1000. After scaling down the cost becomes 2, so the solution matrix will have columns for costs from 0 to 2 for P_1. Similarly, for P_2, P_3, P_4, P_5 and P_6 the corresponding scaling factors are $k_2 = 500$, $k_3 = 333.33333333$, $k_4 = 375$, $k_5 = 400$, $k_6 = 416.66666667$ and the corresponding solution matrices will have 8, 18, 32, 50 and 72 columns.

Table 4.3, 4.4, 4.5 and 4.6 show the solution matrix for P_6 for costs from $0-20$, $21-40$, $41-60$ and $61-72$ respectively. Solution matrix entry ∞ indicates that no such set of items with scaled profit exactly equal to the column number exists with the items in the set containing items from 1 to the row number. After computing the entries in the solution

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
5	∞	26	∞	22	∞	30	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
6	20	26	18	22	26	30	24	17	30	25	28	23	∞	31	∞	27	∞	35	∞	-∞

Table 4.4: Solution matrix for P_6 showing costs from 21-40

	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
6	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Table 4.5: Solution matrix for P_6 showing costs from 41-60

	61	62	63	64	65	66	67	68	69	70	71	72
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
6	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Table 4.6: Solution matrix for P_6 showing costs from 61-72

matrix, the entry corresponding to minimum cost that meets the demand $D = 20$ is selected as the final solution. In this example $S(6,13)$ is the optimal solution.

4.2 Dynamic programming approach to multi-dimensional minimum knapsack

Now we describe a dynamic programming approach to the multi dimensional minimum knapsack problem which can possibly lead to a polynomial time approximation algorithm for this problem. In the multidimensional version of the problem, d dimensions are associated with each item i, the size in dimension j is referred to as $s(i,j)$ for $j \in [1..d]$. The d-dimensional (multidimensional) version has d demands D_j associated with each dimension $j \in [1,...,d]$. The objective is to find a minimum cost subset of items such that the sum of the sizes is greater than the demand in each dimension. Let O be optimal solution, then $\sum_{i \in O} s(i,j) \geq D_j$ for all $j \in [1..d]$ and the cost of items in O is minimum possible.

We begin with a few definitions. Given two subsets of items A and B, we say that A dominates B $(B \subseteq A)$ if in all dimensions the sum of sizes of items in A is greater than or equal to the sum of sizes of items in B, i.e. , for all $j \in [1..d]$ $\sum_{a \in A} s(a,j) \geq \sum_{b \in B} s(b,j)$. In a collection C of subsets of items, we say that a subset B is not dominated if there does not exist any $A \in C$ such that, $B \subseteq A$. A collection of subsets of items in which every subset is non-dominated is called a non dominated collection.

Now we describe the dynamic programming solution to the d dimensional minimum knapsack problem. We assume that the costs are integers. Let $S(i,c)$ be a non-dominated collection of subsets of items from $1..i$ with total costs of items in each subset exactly c. $S(i,c)$ can be computed using the following recurrence,

$$S(i,c) = \begin{cases} S(i-1,c) & \text{if } c_i > c \\ ND(S(i-1,c) \cup_{B \in S(i-1,c-c_i)} \{s_i \cup B\}) & \text{Otherwise} \end{cases}$$

Item number	Cost	Weight(dimension 1)	Weight(dimension 2)
1	1	2	510
2	1	4	508
3	1	8	504
4	1	16	496
5	1	32	480
6	1	64	448
7	1	128	384
8	1	256	256

Table 4.7: Worst-case Knapsack instance

ND is the function that takes a collection of subsets of items and returns a non-dominated collection of subsets of items. Optimal solution is found from the entry $S(n,c)$ with minimum c that contains a feasible subset.

We see that the running time of the above algorithm depends on the running time of the function *ND*. In function *ND*, size of each subset of items in the collection $S(i-1,c)$ in each dimension needs to be compared with the size of each subset of items in the collection $s_i \cup B$ where $B \in S(i-1,c-c_i)$. So, the running time depends on the maximum number of subsets of items in the collection.

4.3 d-dimensional min-knapsack

The algorithm described in section 4.2 can take exponential time in the worst case. We consider a knapsack instance with 2 dimensions and 8 items. The sizes and costs of the items are given in the Table 4.7, and the demands in both the dimensions are 100. In general, the sizes for item i are chosen in increasing order in dimension 1 as powers of two's 2^i and in decreasing order in dimension 2 as $2^{n+1}-2^i$. The costs are chosen as 1 for each item and the scaling factor k is also 1.

Running the algorithm on the instance above we find that the entry $S(8,4)$ in the solution

matrix has the maximum number of subsets of items and this number is 70 which is $\binom{8}{4}$. This implies that all possible subsets with scaled cost 4 are non-dominated. We ran the algorithm for $n \in [5, ..., 20]$ and for each n we found similar results for instances generated. We conjecture that, for even n the maximum number of subsets in a collection is $\binom{n}{n/2}$ and for odd n it is $\binom{n}{\lceil n/2 \rceil}$ or $\binom{n}{\lfloor n/2 \rfloor}$. The entry with maximum number of subsets of items for $n = 8$ is given in Table 4.8 and 4.9.

The algorithm described in Section 4.2 does not imply a polynomial time approximation scheme because the running time of the algorithm can be exponential in the worst-case.

The proof of performance ratio of this algorithm mimics the proof for the FPTAS for one dimensional minimum knapsack described in Section 4.1. If the optimal solution of a multidimensional minimum knapsack instance is represented by the items in the set O and the solution returned by this algorithm is represented by the items in set S then for a fixed ε, this inequality holds, $c(S) \leq c(O)(1+\varepsilon)$. Although the running time of the dynamic program for d-dimensional min-knapsack can be exponential in the worst case but the performance ratio of this algorithm is good when compared with other existing approximation algorithms. This algorithm is an ε-approximation scheme (not polynomial in the worst case) when scaling is performed for a fixed ε.

Items in the Non-dominated collection	Weight(d1)	Weight(d2)
1 2 3 4	30	2018
1 2 3 5	46	2002
1 2 4 5	54	1994
1 3 4 5	58	1990
2 3 4 5	60	1988
1 2 3 6	78	1970
1 2 4 6	86	1962
1 3 4 6	90	1958
2 3 4 6	92	1956
1 2 5 6	102	1946
1 3 5 6	106	1942
2 3 5 6	108	1940
1 4 5 6	114	1934
2 4 5 6	116	1932
3 4 5 6	120	1928
1 2 3 7	142	1906
1 2 4 7	150	1898
1 3 4 7	154	1894
2 3 4 7	156	1892
1 2 5 7	166	1882
1 3 5 7	170	1878
2 3 5 7	172	1876
1 4 5 7	178	1870
2 4 5 7	180	1868
3 4 5 7	184	1864
1 2 6 7	198	1850
1 3 6 7	202	1846
2 3 6 7	204	1844
1 4 6 7	210	1838
2 4 6 7	212	1836
3 4 6 7	216	1832
1 5 6 7	226	1822
2 5 6 7	228	1820
3 5 6 7	232	1816
4 5 6 7	240	1808

Table 4.8: Maximum number of non-dominated subsets for $n = 8$

Items in the Non-dominated collection	Weight(d1)	Weight(d2)
1 2 3 8	270	1778
1 2 4 8	278	1770
1 3 4 8	282	1766
2 3 4 8	284	1764
1 2 5 8	294	1754
1 3 5 8	298	1750
2 3 5 8	300	1748
1 4 5 8	306	1742
2 4 5 8	308	1740
3 4 5 8	312	1736
1 2 6 8	326	1722
1 3 6 8	330	1718
2 3 6 8	332	1716
1 4 6 8	338	1710
2 4 6 8	340	1708
3 4 6 8	344	1704
1 5 6 8	354	1694
2 5 6 8	356	1692
3 5 6 8	360	1688
4 5 6 8	368	1680
1 2 7 8	390	1658
1 3 7 8	394	1654
2 3 7 8	396	1652
1 4 7 8	402	1646
2 4 7 8	404	1644
3 4 7 8	408	1640
1 5 7 8	418	1630
2 5 7 8	420	1628
3 5 7 8	424	1624
4 5 7 8	432	1616
1 6 7 8	450	1598
2 6 7 8	452	1596
3 6 7 8	456	1592
4 6 7 8	464	1584
5 6 7 8	480	1568

Table 4.9: Maximum number of non-dominated subsets for $n = 8$

Chapter 5

Experiments and Results

In this chapter we compare the performance of different approximation algorithms for the minimum knapsack and the multidimensional minimum knapsack problems. Our focus is upon testing the hypothesis that the proposed algorithms have performance ratio better than the theoretical bounds obtained in Chapter 4. We do not attempt to improve the design of the approximation algorithms using the results of the experiments. From a theoretical point of view existence of FPTAS is the best possible algorithmic statement one can hope for; and we do provide a simple FPTAS for the minimum knapsack in Chapter 4. As our target is on showing that although the hypothesis given shows a performance ratio of $3/2$ our algorithms do much better in practice we used the word "experiment" instead of "simulation" in this chapter.

5.1 Test instances

We consider several types of randomly generated instances for experimental analysis. These instances are generated in such a way that they reflect special properties. In almost all of these instances the weights (sizes) are uniformly distributed in a given interval with data range [1,...,R] where R is chosen suitably and profits (costs) are expressed as a function of the weights. The special construction of these instances depends on the function used to compute the profits and this actually defines the special properties associated with each

groups of instances as described in the book by Kellerer *et al.* [24] in chapter 5 section 5.5.

We executed the algorithms on different types of knapsack instances. Although in theory the performance ratio is of our two proposed algorithms is $3/2$ for $\varepsilon = 0.5$, but they show much better performance ratio in practice and is very close to 1 in most cases.

5.1.1 Uncorrelated instances

Profits and weights of the items are selected randomly in the range $[1,...,R]$. There is no correlation between the weight values and the profit values. There can be a large gap between the profit and weight values and hence such instances are generally easy to solve. These type of instances reflect the situation where the profit does not depend on the weight.

5.1.2 Strongly correlated instances

Unlike the uncorrelated instances there is a strong relation between the profit and weight of an item. The weights are distributed in $[1,...,R]$ and profit of item j is defined as $w_j + R/10$. This type of instances correspond to real life situation in which the return is proportional to investment with some fixed charge for each investment. Strongly correlated instances are difficult to solve and hence of particular interest because of two reasons:

- There is a large gap between the optimal integer solution and the optimal fractional solution in the integer programming formulation.
- Looking at the way the profits and weights are defined it is obvious that, sorting the items according to decreasing efficiencies (cost per unit weight) actually corresponds to sorting the items according to weights. This actually makes it difficult to satisfy the capacity constraint of the knapsack problem with equality because for any interval of the ordered items there is a small variation in the weights.

5.1.3 Weakly correlated instances

Weight of an item j, w_j is selected randomly in the range $[1,...,R]$. But unlike the uncorrelated instances the profit of item j, p_j is selected randomly in the range $[w_j - R/10, w_j + R/10]$ such that, $p_j \geq 1$. Although the name suggests that, the relation between profit and weight of an item should not be very strong but actually the weakly correlated instances have a very high correlation between profit and weight of an item and they differ by only a few percent. This actually represents a scenario which is realistic in some sense, i.e. the return of an investment is actually proportional to the invested amount having some small tolerance of variation.

5.1.4 Inverse strongly correlated instances

In these type of instances, profit of item j, p_j is distributed in $[1,...,R]$ and weight w_j is set as $w_j = p_j + R/10$. This is just the same as strongly correlated instances just the relationship between profit and weight is inverse.

5.1.5 Almost strongly correlated instances

These type of instances reflect the properties of both the strongly correlated instances and weakly correlated instances. Weights w_j, are distributed in $[1,...,R]$ and the profits p_j in $[w_j + R/10 - R/500, w_j + R/10 + R/500]$. These type of instances account for some noise added to the fixed charge for each investment.

5.1.6 Subset sum instances

In these type of instances, weight w_j are randomly distributed in $[1,R]$ and $p_j = w_j$. These type of instances reflect the situation where the profit of each item is equal to the weight or proportional to the weight as in the subset sum problem.

5.1.7 Uncorrelated instance with similar weights

In these type of instances weights are distributed in a very narrow range of values such as $[100000, ..., 100100]$ so that the weights of the items are similar and the profits are selected randomly in a wide range of values $[1, ..., 1000]$. Hence, this actually represents uncorrelated instances but the weights are similar.

5.2 Experimental results

Specifications of the machine on which we ran all the experiments are,

Manufacturer: Acer.

Model: Aspire 5920G.

Processor: Intel(R)core(TM)Duo CPU T5550 at 1.83 GHz.

Memory(RAM): 2.0 GB

In the tables showing the performance ratio of the algorithms we have shown up to 12-significant digits after the decimal. The reason behind this is the performance ratio of the proposed algorithms is most of the times very close to 1.

5.2.1 Results for single dimension

In our experiments we set $R = 2^{32} - 1$ the maximum integer value possible in Python. Note that the running time of the pseudo polynomial time algorithm is $O(n^2C)$. Therefore we choose a large C. We compare the primal-dual algorithm [4] for min-knapsack, the proposed FPTAS for min-knapsack, the improved greedy heuristic [7], the proposed ε-approximation scheme for multidimensional minimum knapsack, when $d = 1$ and a gen-

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	1.0214085	1	1.0089257	1	1.009917444
15	1.0467618	1.000305	1	1.000305	1.007861
20	1.04826425	1.0001732	1.0070627	1.0001732	1.0084093
25	1.04473292	1	1.0007502	1	1.0007502
30	1.0259663	1	1.008594	1	1
35	1.0481126	1.0002787	1.014396	1	1.015941
40	1.023532511	1.000002189	1.005573388	1.000002189	1.007032722
45	1.032029064	1.000110539	1.004656089	1.000110539	1.005653528
50	1.027337493	1.000141132	1.006568866	1	1.007619031

Table 5.1: Performance ratio on uncorrelated instances

eralization of [7] mentioned in Chapter 2. We ran the algorithms starting from 10 items in the knapsack instance and increased up to 50 items. We generated 20 different random instances for each n and took the average performance ratio and average running time for these 20 different random instances found by these algorithms. We choose $\varepsilon = \frac{1}{2}$ for the FPTAS and the ε-approximation scheme for the d-dimensional minimum knapsack (for d=1) as the improved greedy heuristic [7] guarantees a performance ratio of $3/2$.

Uncorrelated instances

The algorithms mentioned above were run on uncorrelated knapsack instances. The performance ratios of the algorithms are shown in Table 5.1. The optimal solution used in computation of the performance ratio is found by running an ILPsolver lp_solve [32] on the random instances generated. lp_solve is a free linear (integer) programming solver based on the revised simplex method and the branch-and-bound method. Figure 5.1 graphically represents the performance ratio of the algorithms on uncorrelated knapsack instances.

From Figure 5.1 we can see that the performance ratio of the FPTAS and the ε-approximation scheme is better than the rest of the algorithms. The primal-dual schema has the worst per-

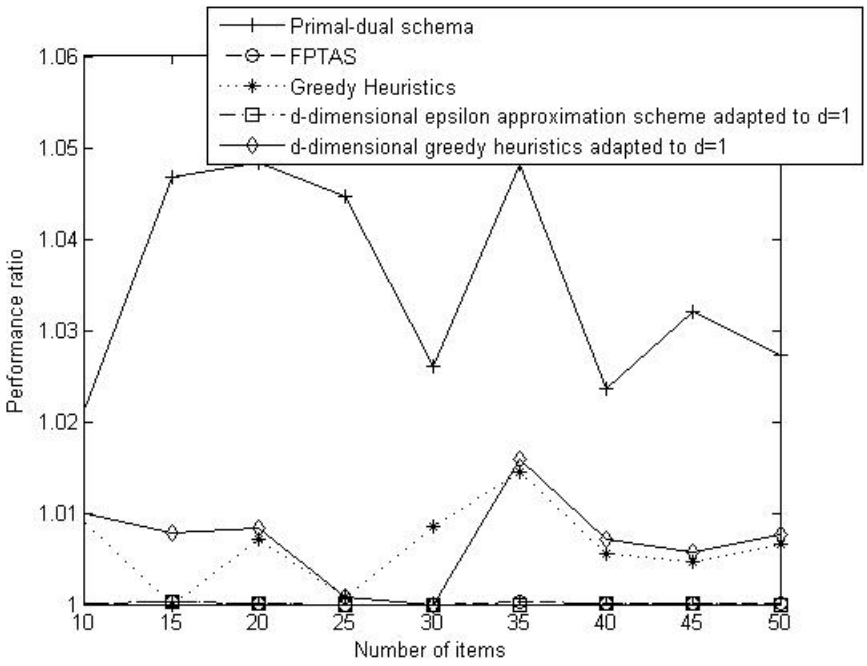

Figure 5.1: Performance ratio on uncorrelated instances

formance ratio among all the algorithms and the greedy heuristics and the generalization of the greedy heuristics have almost the same performance ratio. Note that the worst case performance ratio of the primal-dual algorithm is 2. Therefore these results are not surprising.

Table 5.2 illustrates the running time of the algorithms on uncorrelated instances. From Figure 5.2 it is evident that, the ε-approximation scheme takes much more time than the rest. The FPTAS also takes much more time than the rest of the algorithms. Therefore we take into account only the remaining three algorithms in Figure 5.3. From the figure we note that the generalization of the greedy heuristic takes less time than the other two and the primal-dual schema takes the most time. Next we depict the running time of the FPTAS and ε-approximation scheme as a function of number of items in Figure 5.4 and 5.5.

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	0.05922	0.24626	0.03198	1.00713	0.031256
15	0.07155	0.36763	0.03394	5.23241	0.03919
20	0.08004	0.73481	0.04145	17.6885	0.047225
25	0.10800	1.51266	0.04233	47.1502	0.03029
30	0.15888	2.85938	0.0490	104.6294	0.07494
35	0.18137	5.24168	0.05512	210.0282	0.07789
40	0.25053	9.00787	0.05407	379.8940	0.08339
45	0.30458	15.5480	0.05908	647.3491	0.09616
50	0.41510	23.8145	0.07907	1044.7379	0.10261

Table 5.2: Running time on uncorrelated instances in seconds

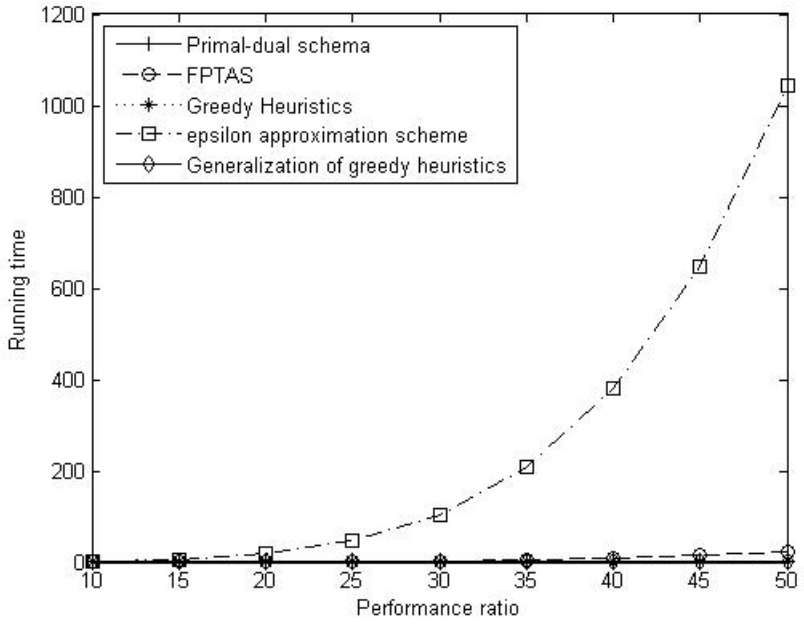

Figure 5.2: Running time (in seconds) of the algorithms on uncorrelated instances

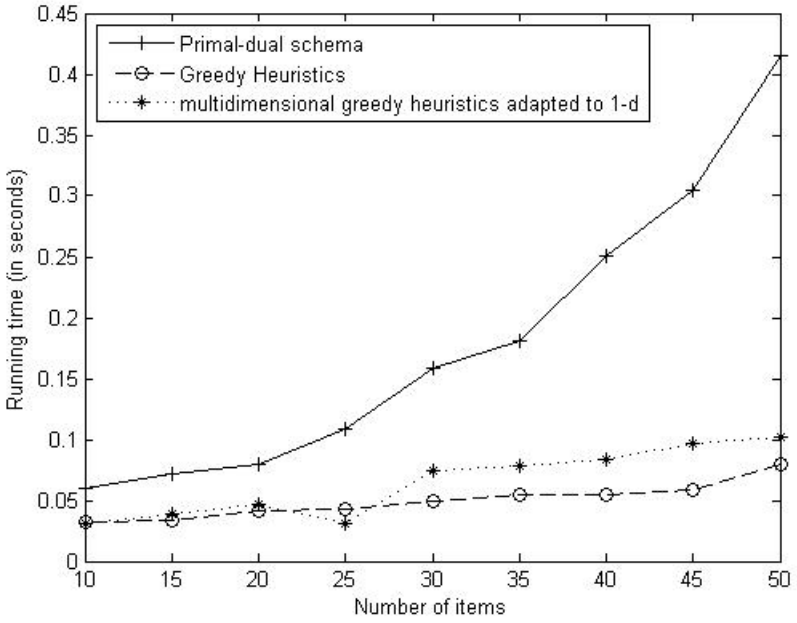

Figure 5.3: Running time (in seconds) of three algorithms on uncorrelated instances

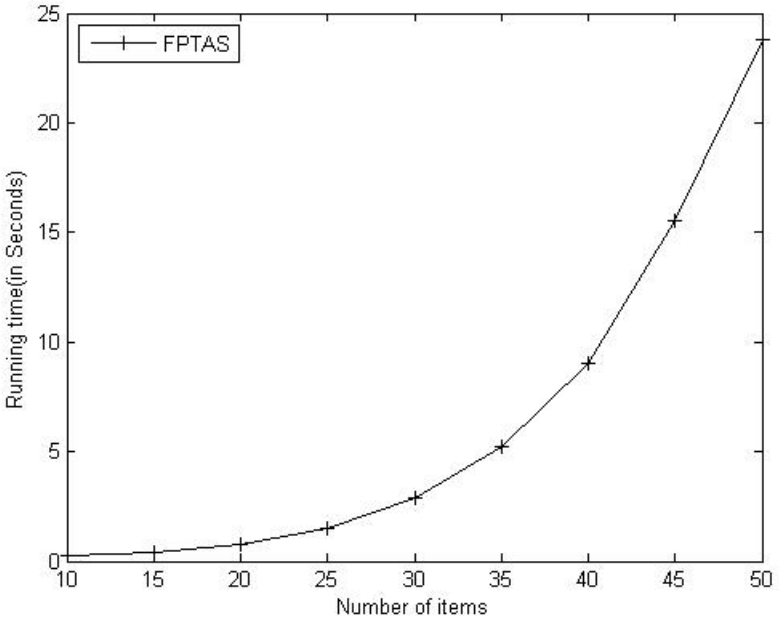

Figure 5.4: Running time of FPTAS
on uncorrelated instances

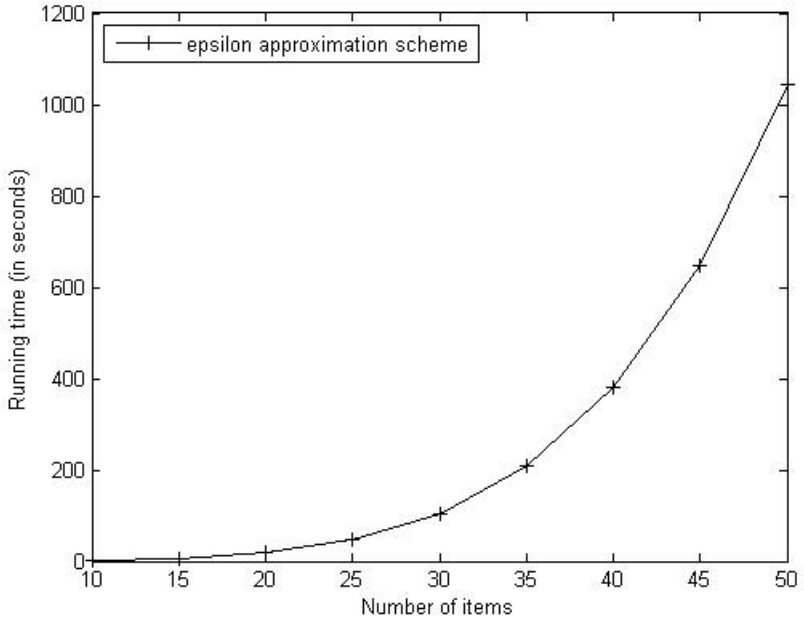

Figure 5.5: Running time of ε-approximation scheme on uncorrelated instances

Strongly correlated instances

We ran the algorithms on strongly correlated knapsack instances with number of items from 10...50 with an interval of 5. The performance ratios of the algorithms for strongly correlated instances are described in Table 5.3.

The performance ratio of the algorithms are portrayed in Figure 5.6. For strongly correlated instances the FPTAS, the ε-approximation scheme and the greedy heuristic has better performance ratio than the primal-dual schema and the generalization of the greedy heuristic algorithm.

The running times of the algorithms are given in Table 5.4. The running times of the three algorithms except for the FPTAS and ε-approximation scheme are shown in Figure 5.7.

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	1.04405	1.00212	1.001679	1.00212	1.01535
15	1.05865	1.002345	1.00239	1.002345	1.00636
20	1.04846	1.001927	1.00091	1.00193	1.0064
25	1.04978	1.00112	1.00075	1.001119	1.0042
30	1.03448	1.0004403	1.0004846	1.000440	1.003892
35	1.032838	1.0005255	1.0004182	1.000525	1.001889
40	1.027074	1.0003316	1.0002949	1.00033	1.0018943
45	1.0244563	1.0003142	1.000254	1.00031419	1.0011333
50	1.027772	1.0002969	1.0000217	1.0002969	1.002234

Table 5.3: Performance ratio on strongly correlated instances

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	0.053813	0.165343	0.03007	0.9729	0.02464
15	0.0548127	0.349001	0.03692	5.11576	0.0316
20	0.064539	0.726873	0.04219	17.1307	0.0266
25	0.077331	1.52555	0.046598	45.175	0.037378
30	0.155341	2.530993	0.0417512	82.72057	0.018468
35	0.1469373	4.476813	0.0365825	161.82416	0.01780
40	0.1993809	7.6731439	0.050140	307.7466	0.0250
45	0.2826214	12.12999	0.0550649	512.69458	0.031298
50	0.3724166	18.092079	0.063097	774.128	0.037356

Table 5.4: Running time on strongly correlated instances in seconds

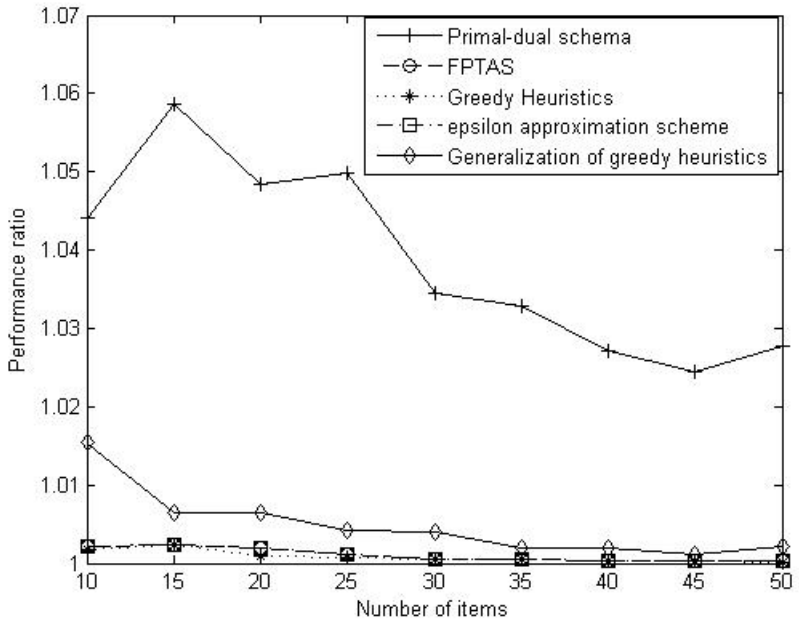

Figure 5.6: Performance ratio of the algorithms on strongly correlated instances

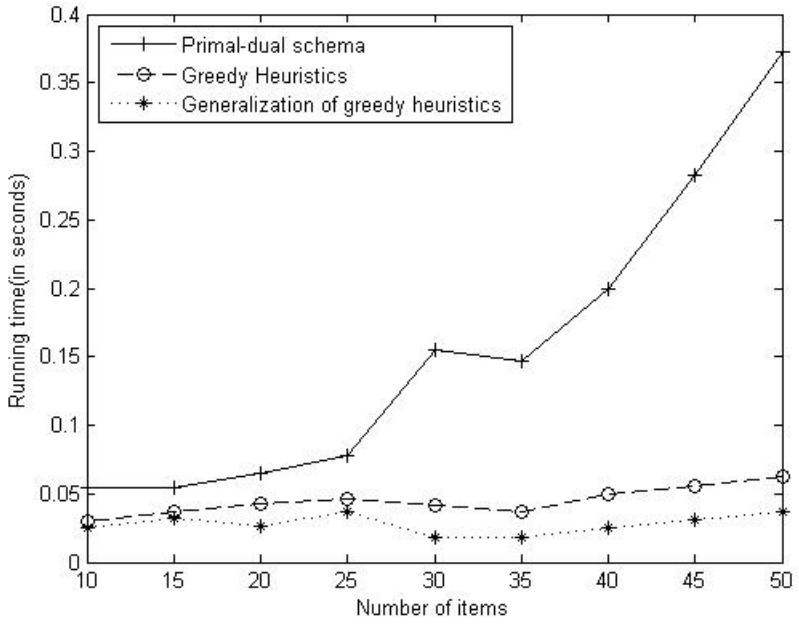

Figure 5.7: Running time of three algorithms on strongly correlated instances

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	1.105400	1.0030061	1.0094406	1.002258	1.0237269
15	1.0567122	1.0005357	1.0119130	1.0005357	1.0176041
20	1.0439875	1.0005378	1.0063693	1.0004203	1.0106831
25	1.050380	1.000690	1.0034388	1.000610	1.0065710
30	1.031176	1.0005957	1.0024008	1.000595	1.0039990
35	1.0320641	1.000213	1.0019481	1.0001592	1.0034831
40	1.0288530	1.00014475	1.0014521	1.0001382	1.0028035
45	1.023025	1.0001108	1.0012939	1.0000449	1.0024337
50	1.018144	1.0000727	1.0008261	1.0000727	1.002555

Table 5.5: Performance ratio on Weakly Correlated instances

Weakly correlated instances

Like the previous two types of knapsack instances we ran experiments on the weakly correlated instances and the results are given in Table 5.5 and Table 5.6. Figure 5.8 shows the performance ratio of the algorithms on weakly correlated instances. We see from the figure that the performance ratio of the FPTAS and the ε-approximation scheme is better than the remaining three algorithms. Primal-dual schema has the worst performance ratio among all the algorithms. The greedy heuristic and the generalization of the greedy heuristic has almost the same performance ratio but the greedy heuristic always performs better than the generalization of the greedy heuristic.

Running time of the three algorithms except the FPTAS and the ε-approximation scheme are shown in Figure 5.9. As for previous two types of instances the primal-dual schema takes more time than the other two and other two algorithms have running time very close to each other.

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	0.0518565	0.17803	0.0279147	0.866330	0.0069038
15	0.0628840	0.3378578	0.028594	4.520310	0.00839441
20	0.0903866	0.7019758	0.029996	15.16238	0.010213
25	0.160394	1.4150664	0.0354767	39.42007	0.014554
30	0.2032163	2.6055073	0.0347604	85.55213	0.017715
35	0.3158113	4.580012	0.040072	167.694	0.021273
40	0.4540825	7.569181	0.043116	312.4766	0.025826
45	0.684828	12.2286	0.045197	515.0644	0.0318377
50	0.966278	18.158447	0.04551	802.7516	0.038199

Table 5.6: Running time on Weakly Correlated instances in seconds

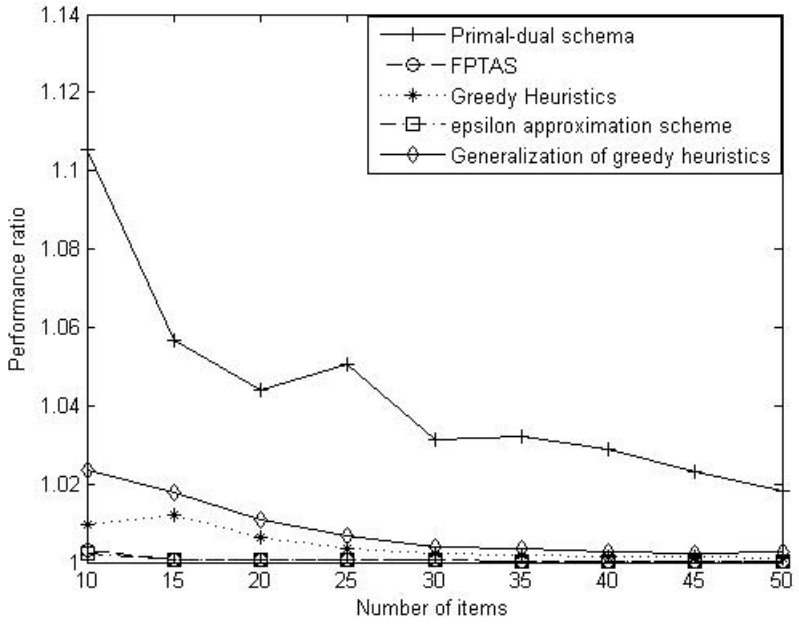

Figure 5.8: Performance ratio of the algorithms on weakly correlated instances

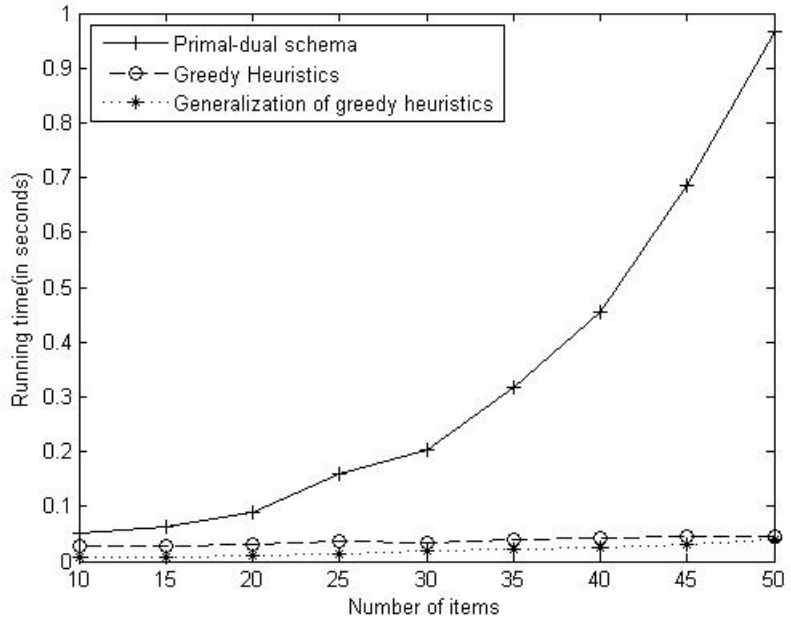

Figure 5.9: Running time of the three algorithms on weakly correlated instances

Inverse strongly correlated instances

We ran all the five algorithms on the randomly generated instances and the performance ratio is shown in Table 5.7 and the running time is shown in Table 5.8.

Figure 5.10 shows the performance ratio of the algorithms in graphical form. The performance ratio of the greedy heuristic, FPTAS and ε-approximation scheme are almost the same. Here also the primal-dual schema has the worst performance ratio.

Figure 5.11 shows the running time of the three algorithms except for the FPTAS and the ε-approximation scheme.

Almost strongly correlated instances

The results obtained for almost strongly correlated instances are given in tabular form in Tables 5.9 and 5.10. Table 5.9 shows the performance ratios of the algorithms and Table

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	1.19089354	1.00179457	1.0259339	1.0017945	1.0259339
15	1.07962556	1.00039226	1.0242405	1.000392	1.0242405
20	1.06360413	1.00039636	1.0164269	1.0003963	1.016426
25	1.05519219	1.000255539	1.0102269	1.0002555	1.0102269
30	1.03701317	1.0002492	1.0093071	1.0002492	1.0093071
35	1.03624255	1.0000997	1.0093940	1.000099	1.0093940
40	1.0350417	1.00008978	1.0059404	1.000089	1.0059404
45	1.0282799	1.00005321	1.0096228	1.0000532	1.0096228
50	1.0254551	1.00003974	1.0051974	1.0000397	1.0051974

Table 5.7: Performance ratio on inverse strongly correlated instances

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	0.048304	0.162335	0.025639	0.85525	0.0078458
15	0.063219	0.337045	0.024402	4.47611	0.00771719
20	0.135223	0.73776	0.026426	14.9686	0.00926311
25	0.163696	1.4112	0.0284767	38.94518	0.0257644
30	0.27077	2.656175	0.02874	85.218174	0.017664
35	0.4313	4.6335	0.031114	168.5598	0.0207372
40	0.6363	7.55819	0.0346442	302.860	0.0258261
45	0.9306	11.6562	0.029507	512.051	0.0291143
50	1.40406	17.82194	0.038023	839.537	0.0391939

Table 5.8: Running time on inverse strongly correlated instances in seconds

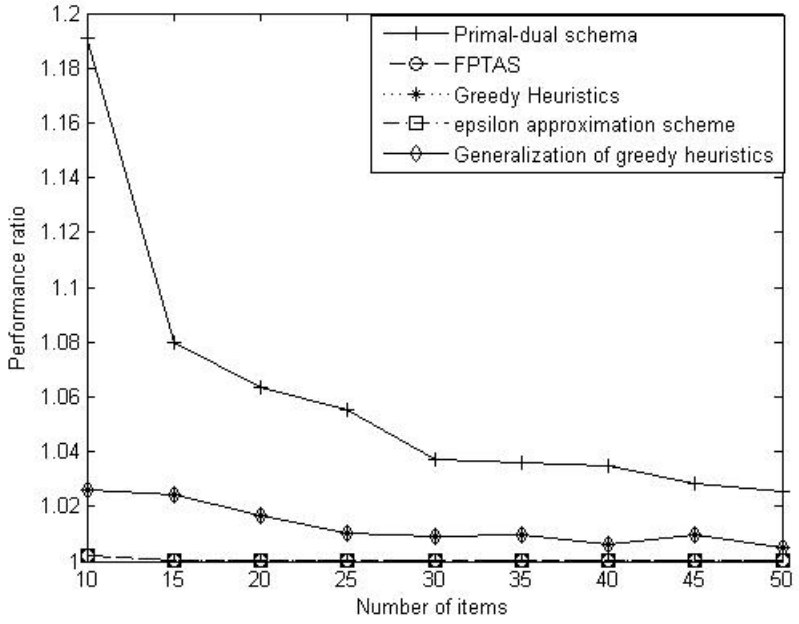

Figure 5.10: Performance ratio of the algorithms on inverse strongly correlated instances

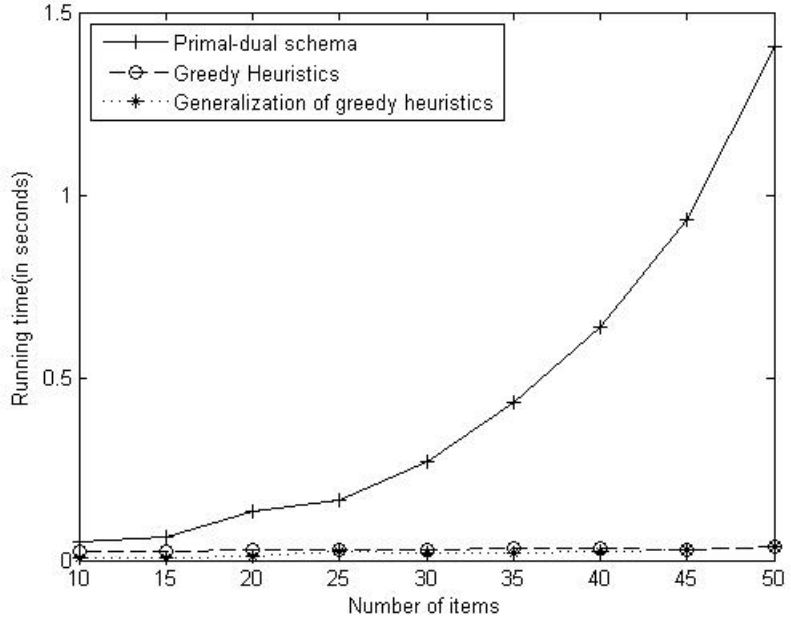

Figure 5.11: Running time of the three algorithms on inverse strongly correlated instances

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	1.075260111	1.00307902	1.00364028	1.00307902	1.01671998
15	1.07660228	1.00253952	1.00232569	1.00253952	1.01030349
20	1.03560679	1.0012162	1.00072765	1.0012162	1.00525749
25	1.05115084	1.00076863	1.00100424	1.00076863	1.00396752
30	1.03100549	1.00090515	1.00079624	1.00090515	1.0035117
35	1.02968065	1.00053406	1.00064730	1.00053406	1.00188665
40	1.03509500	1.00063318	1.00041958	1.00063318	1.00197419
45	1.0259094	1.00051279	1.00033003	1.00051279	1.00154668
50	1.02008338	1.00036782	1.00032525	1.00036782	1.0014549

Table 5.9: Performance ratio on almost strongly correlated instances

5.10 shows the running times of the algorithms.

Figure 5.12 shows the performance ratio for the algorithms in graphical format. The performance ratio of the primal dual algorithm is the worst and the FPTAS, the ε-approximation scheme and greedy heuristic algorithm have almost the same performance ratio.

From Figure 5.13 we find that the greedy heuristic takes least amount of time and the primal-dual schema takes the most time of the three algorithms.

5.2.2 Results for multiple dimensions

We compared two algorithms for the multidimensional min-knapsack problem. The first one is the ε-approximation scheme proposed in Chapter 4 for $\varepsilon = 0.5$ and another one is the generalization of the greedy heuristic algorithm in Chapter 2. We performed experiments on the different types of instances as for the single dimension. We used the ILPsolver lp_solve to find out the optimal solution to compute the performance ratio of the algorithms. We generate 20 different random instances for each d and took the average performance ratio and the average running time for these 20 different instances found by these two

n	Primal-Dual algorithm	FPTAS	Greedy Heuristic	ε-approximation scheme for $d = 1$	Greedy Heuristic for $d = 1$
10	0.04686	0.13942	0.02609	0.8335	0.00717
15	0.05078	0.2755	0.02765	4.33261	0.00802
20	0.06156	0.6468	0.0297	14.4306	0.00921
25	0.07603	1.3031	0.0341	37.2163	0.0138
30	0.1368	2.4850	0.0373	82.16698	0.0165
35	0.1589	4.4515	0.04047	159.16958	0.01961
40	0.2010	7.39227	0.0456	286.07545	0.02419
45	0.2941	11.6684	0.0526	485.84	0.029407
50	0.3745	17.1643	0.0548	775.99686	0.03816

Table 5.10: Running time on almost strongly correlated instances in seconds

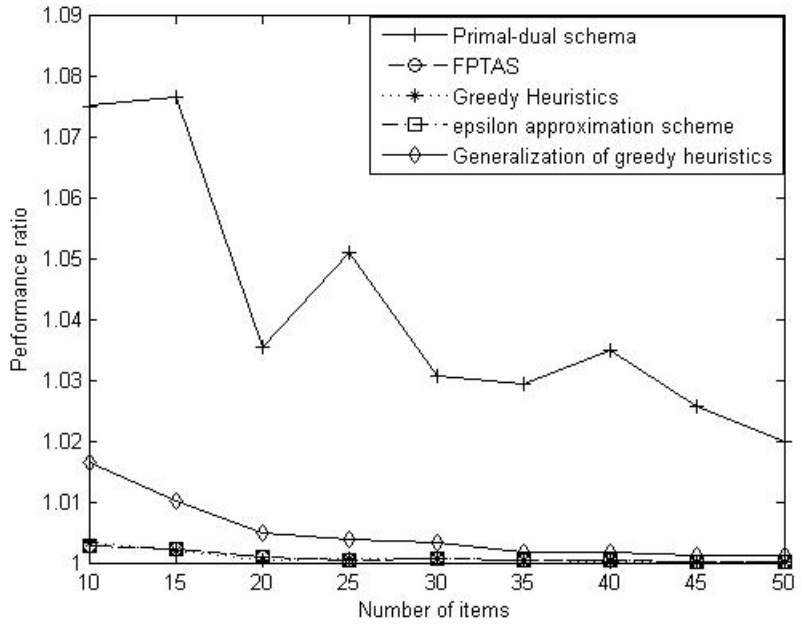

Figure 5.12: Performance ratio of the algorithms on almost strongly correlated instances

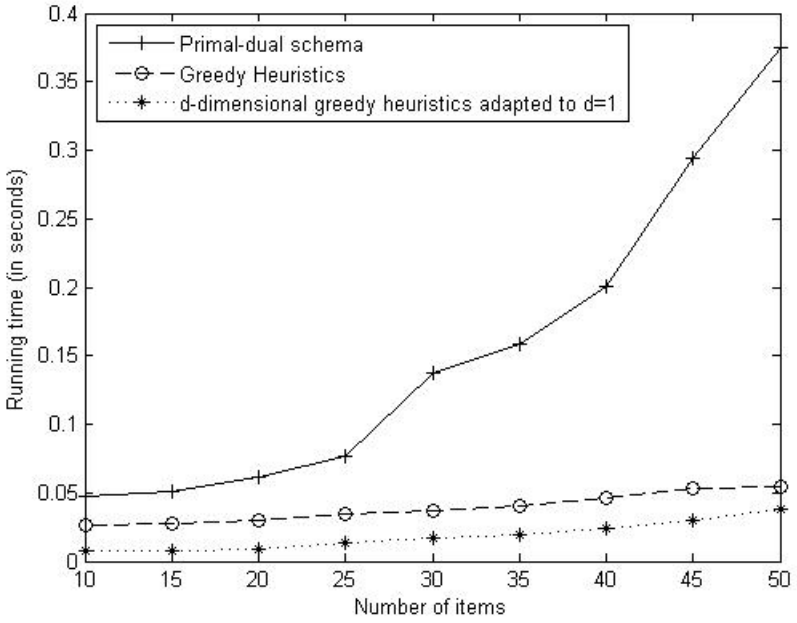

Figure 5.13: Running time of the three algorithms on almost strongly correlated instances

algorithms. Without loss of generality demands in each of the dimensions are chosen to be the same.

Uncorrelated instances

The two algorithms were run for 15 items for dimensions $2, 4, 6, ..., 20$. The corresponding results are given in Table 5.11. Figure 5.14 shows the running time of the algorithms graphically for $n = 15$. Figure 5.15 shows the performance ratio of the two algorithms for increasing dimensions for a fixed value of $n = 15$.

Strongly correlated instances

For the multidimensional case the profit of an item in a strongly correlated instance is generated by taking the average of all the weights of the item in different dimensions and then using the formula described at the beginning of the chapter.

d	ε-approximation scheme (running time in seconds)	ε-approximation scheme (performance ratio)	Generalization of greedy heuristics (running time in seconds)	Generalization of greedy heuristics (performace ratio)
2	5.94990	1.0	0.009041	1.03967278224
4	9.40856	1.0	0.012308	1.09730297248
6	14.72312	1.0	0.013530	1.08632680237
8	17.24813	1.0	0.015775	1.12600757735
10	25.60311	1.0	0.017528	1.12948821425
12	25.70233	1.0	0.019245	1.11523576012
14	37.64886	1.0	0.020880	1.11903428197
16	41.62675	1.00025542356	0.022962	1.08348162442
18	53.61691	1.0	0.026398	1.0767496281
20	57.37521	1.00009998642	0.027571	1.08480945647

Table 5.11: Results on uncorrelated instances for multiple dimensions for $n = 15$

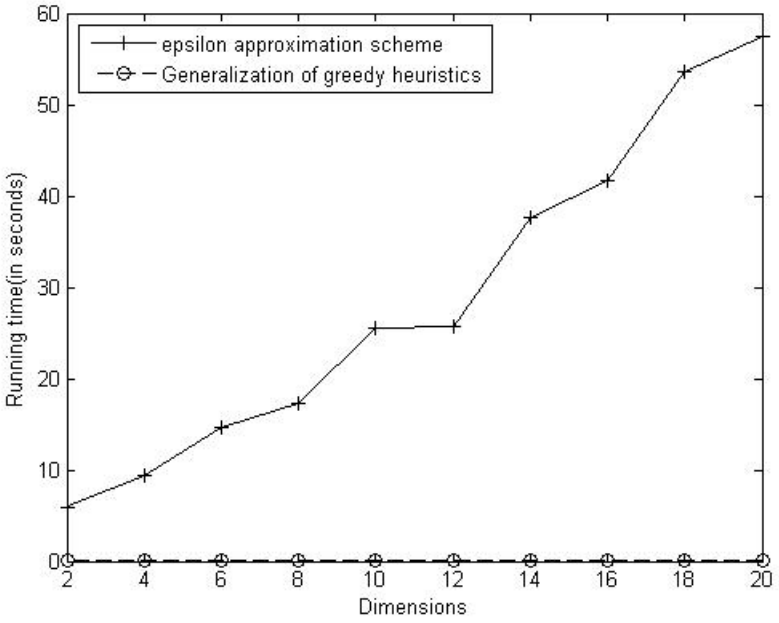

Figure 5.14: Running time of the two algorithms on uncorrelated instances for $n = 15$

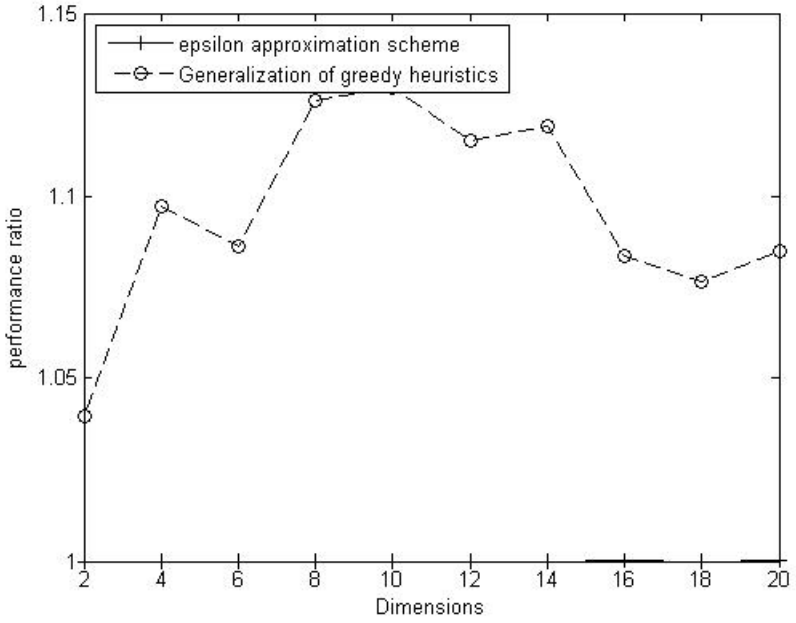

Figure 5.15: Performance ratio of the two algorithms on uncorrelated instances for $n = 15$

We ran experiments on strongly correlated instances for $n = 15$ and we increase the dimensions in each case from 2 to 20 in increments of 2. The results are shown in Table 5.12.

Figures 5.16 and 5.17 show graphically the performance ratio and running time respectively for the two algorithms for $n = 15$.

Weakly correlated instances

The average weight of an item is calculated as in the case of strongly correlated instances and profits are calculated according to the formula mentioned at the beginning of the chapter.

We show the results for increasing dimensions from 2 to 20 in Table 5.13 for the two algorithms. Figures 5.18 and 5.19 show the performance ratio and running time of the

d	ε-approximation scheme (running time in seconds)	ε-approximation scheme (performance ratio)	Generalization of greedy heuristics (running time in seconds)	Generalization of greedy heuristics (performace ratio)
2	11.327893	1.00117908998	0.0092239	1.03490662656
4	40.650655	1.00095496191	0.011524	1.13241765442
6	63.179711	1.001645378158	0.0116852	1.184287290504
8	107.4764	1.0	0.013782	1.21517808473
10	240.29170	1.00022137748	0.014782	1.14843576936
12	253.34931	1.00126386655	0.015752	1.15515255464
14	435.62169	1.0	0.017534	1.17173047971
16	382.11774	1.0	0.019044	1.29110291048
18	1510.1381	1.0	0.020594	1.21660112724
20	813.01332	1.00142724213	0.020739	1.18741834605

Table 5.12: Results on strongly correlated instances for multiple dimensions for $n = 15$

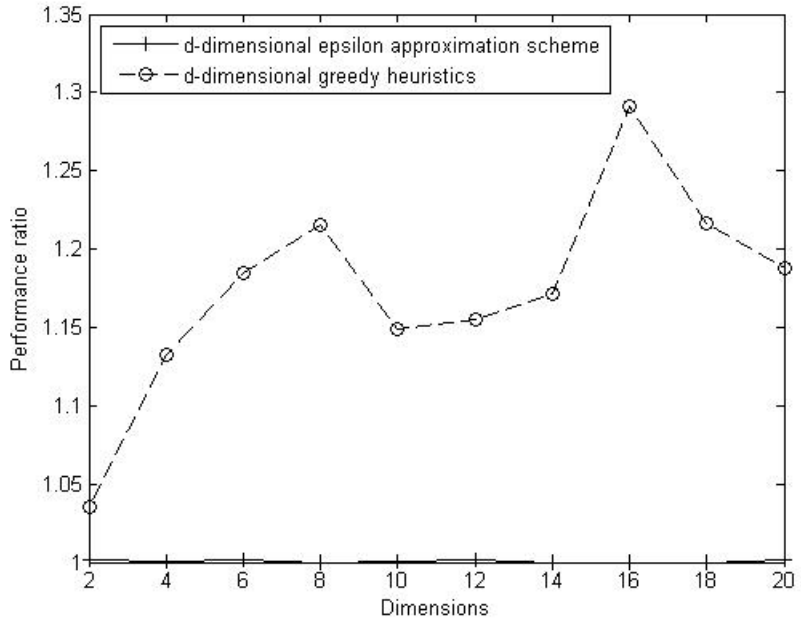

Figure 5.16: Performance ratio of the two algorithms on strongly correlated instances for $n = 15$

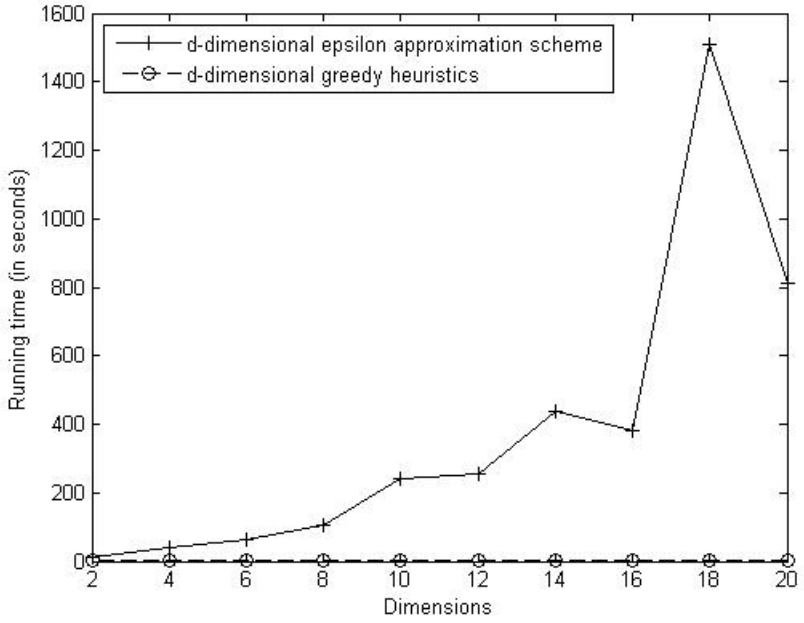

Figure 5.17: Running time of the two algorithms on strongly correlated instances for $n = 15$

d	ε-approximation scheme (running time in seconds)	ε-approximation scheme (performance ratio)	Generalization of greedy heuristics (running time in seconds)	Generalization of greedy heuristics (performace ratio)
2	8.55358	1.0	0.0090353	1.09775008374
4	28.24180	1.00160023788	0.0098139	1.11434947822
6	46.54911	1.00035996417	0.011021	1.14466710799
8	69.21992	1.00032664304	0.012934	1.18030618906
10	74.52352	1.0	0.014438	1.15396819017
12	84.21646	1.00122003248	0.016209	1.1244302272
14	123.68793	1.0003767695	0.017591	1.13131401434
16	124.03993	1.0	0.019942	1.13867798046
18	146.39304	1.00010630403	0.021416	1.19417842224
20	188.39408	1.00079467176	0.02239	1.15520633528

Table 5.13: Results on weakly correlated instances for multiple dimensions for $n = 15$

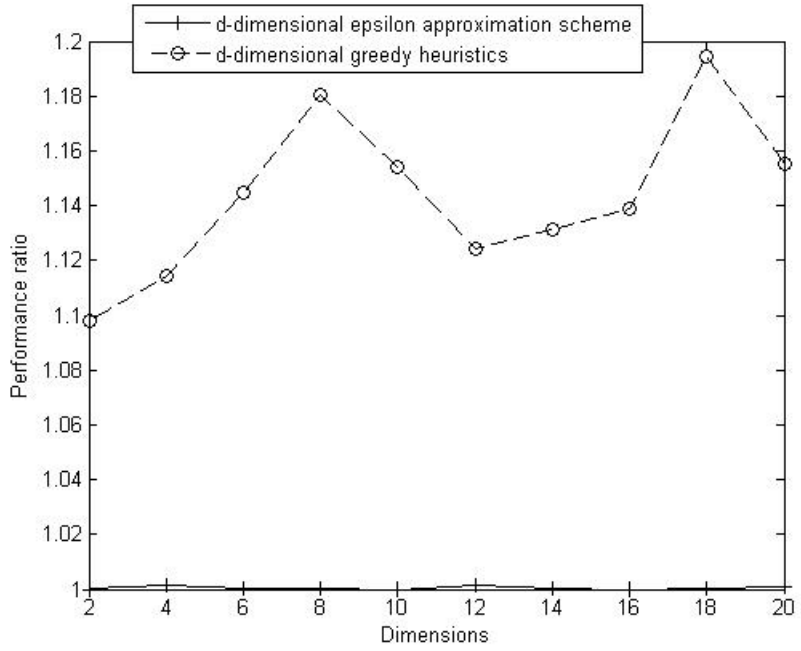

Figure 5.18: Performance ratio of the two algorithms on weakly correlated instances for $n = 15$

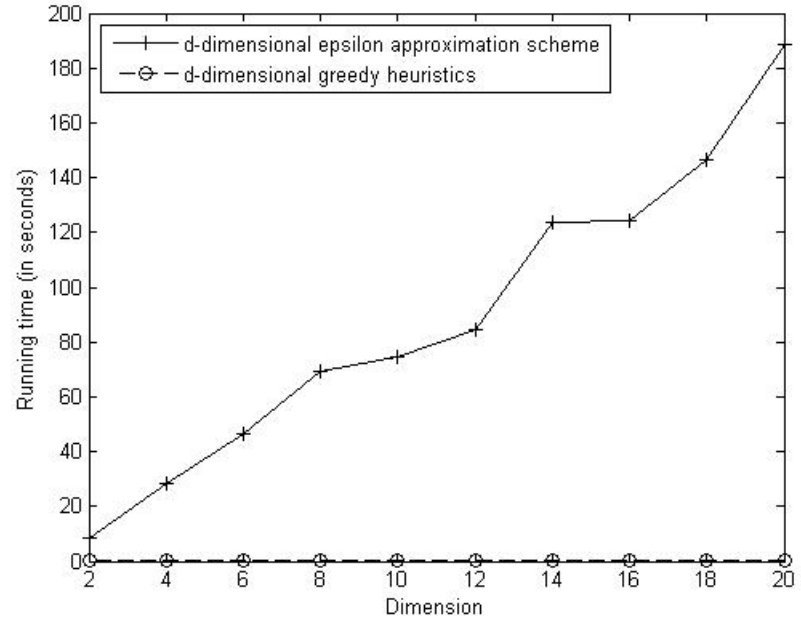

Figure 5.19: Running time of the two algorithms on weakly correlated instances for $n = 15$

d	ε-approximation scheme (running time in seconds)	ε-approximation scheme (performance ratio)	Generalization of greedy heuristics (running time in seconds)	Generalization of greedy heuristics (performace ratio)
2	10.57553	1.00189629567	0.008047	1.05201577442
4	40.700069	1.00136283812	0.0091161	1.11418994564
6	75.79224	1.00096660053	0.0113100	1.12927590362
8	95.08277	1.00034415469	0.010879	1.15549634454
10	180.7887	1.0011801543	0.0135901	1.19557553838
12	217.3170	1.00102186655	0.0167371	1.18971033601
14	540.1390	1.00037951951	0.017688	1.27092129829
16	585.7146	1.00093040115	0.0192328	1.18742701249
18	1555.2405	1.00069750383	0.0211130	1.21955912874
20	1500.26419	1.0	0.022132	1.22410586893

Table 5.14: Results on almost strongly correlated instances for multiple dimensions for $n = 15$

algorithms graphically.

Almost strongly correlated instances

For almost strongly correlated instances the average weight is calculated the same way as in the case of strongly correlated instances and profits are calculated according the previous formula.

Table 5.14 shows the running time and performance ratio of the two algorithms for $n = 15$ and increasing dimensions from 2 to 20. Figures 5.20 and 5.21 show the results graphically.

5.2.3 Performance of the greedy heuristics for d dimensions

The performance of the greedy heuristics is compared with the ILP solver lp_solve in terms of running time for different dimensions and number of items. The goal was to find out the running time for number of items as large as possible for a dimension. Because of time constraint we waited for a program to finish a reasonable time otherwise interrupt it and try

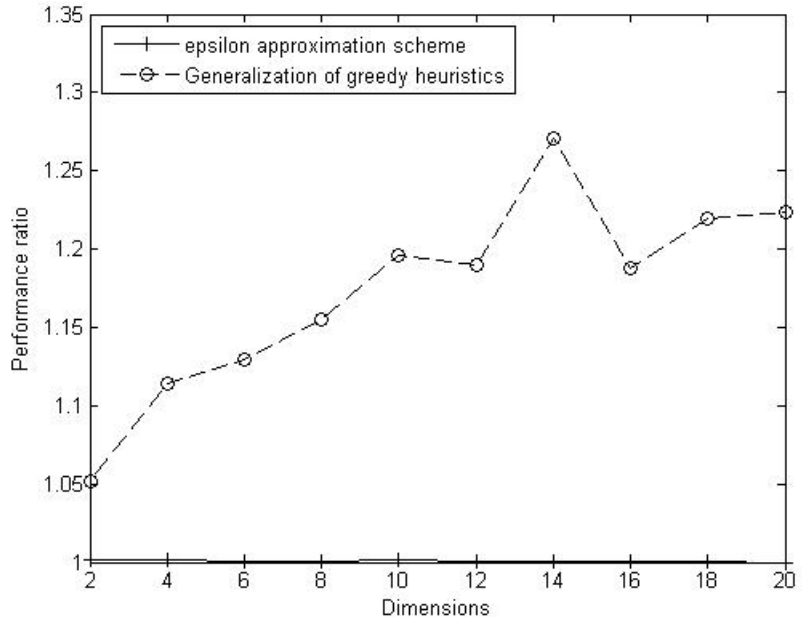

Figure 5.20: Performance ratio of the two algorithms on almost strongly correlated instances for $n = 15$

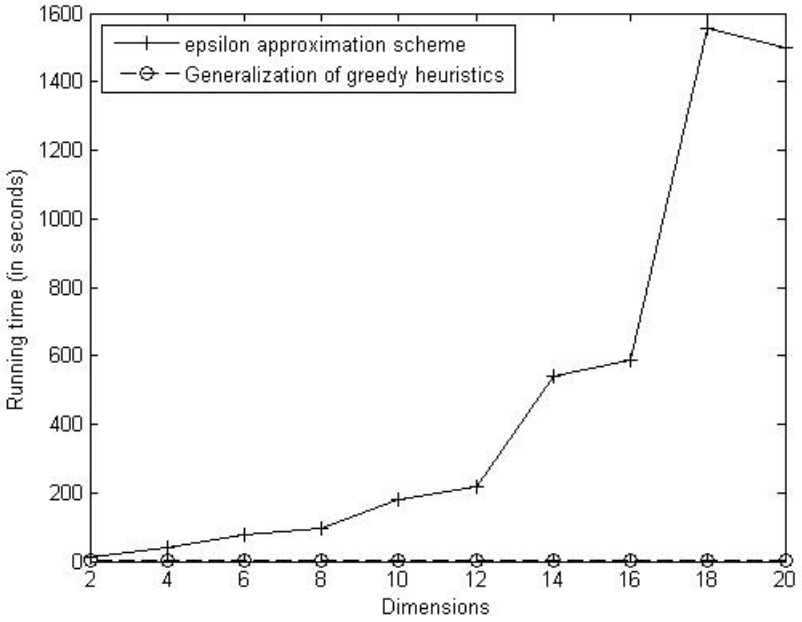

Figure 5.21: Running time of the two algorithms on almost strongly correlated instances for $n = 15$

d	n	Generalized greedy heuristic for d dimensions (Running time in seconds)	Integer Program (Running time in seconds)	Performance ratio of generalized greedy heuristic for d dimensions
2	1000	249.28994	270.91968	1.03147646045
4	500	68.790903	828.444937	1.04972680397
6	300	22.623653	221.70324	1.05937558994
8	200	9.9643800	90.013945	1.07983553283
10	150	5.2726227	10.303540	1.11705055541
15	150	8.1468169	19.879596	1.09774677049
20	150	10.548375	36.425804	1.07968844346
30	150	16.523595	92.83838	1.13439421593
40	150	21.598281	108.68777	1.10743181544
50	150	27.349385	334.10194	1.105094458

Table 5.15: Results on uncorrelated instances for multiple dimensions

the next run of the program for a smaller n. We took the average of 10 different random instances for finding the average running time and average performance ratio.

Uncorrelated instances

The instances are generated the same way they were generated in Subsection 5.2.2. The time taken for the uncorrelated instances by the greedy heuristics for d- dimensions and the ILP solver lp_solve is given in Table 5.15. The performance ratio of the greedy heuristic is also provided in the table.

Strongly correlated instances

The instances are generated the same way as it was in the previous section. Table 5.16 shows the corresponding results for the greedy heuristic and the lp_solve.

d	n	Generalized greedy heuristic for d dimensions (Running time in seconds)	Integer Program (Running time in seconds)	Performance ratio of generalized greedy heuristic for d dimensions
2	100	0.274930	5756.4464	1.0228555239
4	80	0.3964825	7809.7019	1.0745573213964
6	60	0.280935	2628.4652	1.10836674755
8	50	0.335798	1869.4573	1.13501826196
10	50	0.279946	1747.4059	1.13410882606
15	50	0.431142	2195.0220	1.15885225275
20	50	0.552069	5691.1679	1.15851261959
30	50	0.707125	4132.4788	1.16509917036
40	50	0.969720	7347.6498	1.14655076293
50	45	0.909189	2434.4979	1.16031082065

Table 5.16: Results on strongly correlated instances for multiple dimensions

Weakly correlated instances

Results on the weakly correlated instances are given in Table 5.17. It shows the performance ratio and the running time of the greedy heuristics as well as the running time of the lp_solve.

Almost strongly correlated instances

Running time of the greedy heuristics and the lp_solve is given in Table 5.18. The corresponding performance ratio is also provided.

5.2.4 Improvement of the ε-approximation scheme for $d = 2$

In a non-dominated collection of items for 2 dimensions if the items are ordered non-decreasingly according to their weights in dimension 1, the items would be ordered in the reverse order for the weights in dimensions 2 and this property makes it easier for the calculation of the non-dominated collection of items. We first order the items in non-decreasing

d	n	Generalized greedy heuristic for d dimensions (Running time in seconds)	Integer Program (Running time in seconds)	Performance ratio of generalized greedy heuristic for d dimensions
2	500	37.77000	1278.84156	1.02093457443
4	200	9.557667	3689.50923	1.094602430407618
6	100	0.97288	2416.85488	1.046707478517
8	70	0.50017	839.964289	1.15389569988
10	50	0.25822	46.512052	1.12523010869
15	50	0.39201	60.235381	1.10690544074
20	50	0.53534	96.75626	1.20229041948
30	50	0.73568	173.40676	1.11161402051
40	50	0.95361	319.56722	1.09375161017
50	45	1.21252	506.04430	1.13989627343

Table 5.17: Results on weakly correlated instances for multiple dimensions

d	n	Generalized greedy heuristic for d dimensions (Running time in seconds)	Integer Program (Running time in seconds)	Performance ratio of generalized greedy heuristic for d dimensions
2	100	0.2866768	6164.65640	1.01408637103
4	80	0.308048	32118.1221	1.04908466357
6	50	0.1424694	581.152882	1.10569549972
8	50	0.2058156	2956.31721	1.14851901233
10	50	0.2414775	4706.66022	1.14710477867
15	50	0.3764509	2854.20271	1.18848383538
20	50	0.529636	1427.31194	1.13105046977
30	50	0.7682250	2501.28726	1.20454039487
40	50	0.9487125	3381.95086	1.15792076689

Table 5.18: Results on almost strongly correlated instances for multiple dimensions

n	Running time in seconds (Before improvement)	Running time in seconds (After improvement)
10	1.192147	1.383406
15	13.97594	12.64641
20	79.74382	55.18318
25	298.45284	169.4196
30	2526.76194	743.60619
35	37009.1769	4463.6875

Table 5.19: Improvement in running time of the ε-approximation scheme for $d = 2$

order and then find the insertion position for the new item in the ordered collection in both the dimensions using binary search. From the position of the item in both the dimensions it is trivial to determine whether this item is dominated by any of the existing items in the collection or one or more items already in the collection are dominated by this new item or not. If the sum of the insertion positions of the new item is greater than the number of items after insertion then this new item is non-dominated. Otherwise this item is dominated by at least one of the existing non-dominated items in the collection. This technique makes the calculation of non-dominated set of items much faster as binary searching and sorting takes $O(n \log n)$ time where checking against each item takes $O(n^2)$ time. However this approach does not generalize for $d \geq 3$. The results after improvement is given in Table 5.19.

5.3 Concluding remarks

In this chapter we experimentally analyze the performance ratio and the running time of the two algorithms proposed in Chapter 4, along with three other algorithms, on different types of knapsack instances. We find that our two proposed algorithms have better performance ratio than other known algorithms for most of the knapsack instance types but they run slower than all the other algorithms. In the experiments we set $\varepsilon = 0.5$ but in practice the

two proposed algorithm show performance ratio very close to 1 instead of the theoretical bound of $3/2$.

In the thesis, we give an FPTAS for the 0-1 min knapsack problem. We also give an ε-approximation scheme for the multidimensional min knapsack problem based on a generalization of the dynamic programming and scaling.

The proposed FPTAS and the proposed ε-approximation scheme have the same performance ratio for most instances. We empirically show that, for $\varepsilon = 0.5$ the FPTAS gives a good performance ratio compared to the other algorithms though it takes more time than the other algorithms except for the ε-approximation scheme. On the contrary, the $3/2$ approximation greedy algorithm takes the least amount of time for most cases but does not have a good performance ratio compared to the FPTAS when $\varepsilon = 0.5$. The primal-dual schema based algorithm has an approximation ratio of 2 and the running time is more than the greedy approximation algorithm and less than the FPTAS. The experimental performance ratio of the primal-dual schema based algorithm is the worst among the compared algorithms.

We also compare two algorithms for the multidimensional knapsack problem. The dynamic programming based approximation scheme gives a better performance ratio than the greedy approach but the greedy method takes much less time than the dynamic programming based approach which in the worst case is not polynomial.

We compare the running time of the greedy approach to that of the ILP solver lp_solve in section 5.2.3. We observe that, for uncorrelated instances unless $n = 1000$ and $d = 2$ the lp_solve gives the optimal result in a reasonable time but the greedy approach is always faster and has good performance ratio. For strongly correlated and almost strongly correlated instances the ILP solver lp_solve takes much more time than the greedy ap-

proach. lp_solve runs faster on weakly correlated instances rather than on strongly and almost strongly correlated instances and it runs fastest on uncorrelated instances.

Chapter 6

Conclusion and Future Work

In this thesis we described some previous works on different types of knapsack problems in Chapter 2. In Chapter 3 the MITACS internship project was discussed in detail and how this project was related to the knapsack problem is also described. The objective of this internship program was to reduce the cost associated with the fleet used by of DIAGEO, a client of Canadian Pacific Logistics Solutions.

In Chapter 4 we proposed a scaling based FPTAS for the minimum knapsack problem and a scaling and dynamic programming based ε-approximation scheme for the multidimensional minimum knapsack problem. We experimentally analyzed the performance of these two algorithms with respect to some other existing algorithms on different types of knapsack instances in Chapter 5.

The proposed ε-approximation scheme for the multidimensional minimum knapsack takes longer time to execute although it gives a good performance ratio. Two questions of immense interest that remain unanswered in this thesis are:

- The conjecture in Chapter 5 on the exact number of non-dominated solutions in the worst case instances proposed.
- Modification to the *ND* function so as to consider only a selected number of subsets

of items satisfying the non-dominance criteria. Also the computation of the non-dominated set of solutions can be made efficient by using different data structures.

Bibliography

[1] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Rsearch*, 28:1130–1154, 1980.

[2] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[4] T. Carnes and D. B. Shmoys. Primal-dual schema for capacitated covering problems. In *IPCO*, pages 288–302, 2008.

[5] R. D. Carr, L. K. Fleischer, V. J. Leung, and C. A. Phillips. Strengthening integrality gaps for capacitated network design and covering problems. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 106–115, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[6] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[7] J. Csirik, J. B. G. Frenk, M. Labbé, and S. Zhang. Heuristics for the 0-1 min-knapsack problem. *Acta Cybernetica*, 10(1-2):15–20, 1991.

[8] G. Dobson. Worst-case analysis of greedy heuristics for integer programming with nonnegative data. *Mathematics of Operations Research*, 7:515–531, 1982.

[9] D. Dor and U. Zwick. Selecting the median. *SIAM Journal on Computing*, 28:1722–1758, 1999.

[10] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.

[11] G. E. Fox and G. D. Scudder. A heuristic with tie breaking for certain 0-1 integer programming models. *Naval Research Logistics Quarterly*, 32:613–623, 1985.

[12] A. Fréville. The multidimensional 0-1 knapsack problem: An overview. *European Journal of Operational Resarch*, 155:1–21, 2004.

[13] A. M. Friese and M. R. B. Clarke. Approximation algorithms for the *m*-dimensional 0-1 knapsack problem: worst-case and probabilistic analysis. *European Journal of Operational Research*, 15:100–109, 1984.

[14] E. V. Levner G. V. Gens. Computational complexity of approximation algorithms for combinatorial problems. *Lecture Notes in Computer Science*, 74:292–300, 1979.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the theory of NP-Completeness*. Pearson Addison Wesley, 1979.

[16] D. Gaur. Personal Communication.

[17] B. Gavish and H. Pirkul. Allocation of databases and processors in a distributed computing system. *Management of Distributed Data Processing*, pages 215–231, 1982.

[18] P. C. Gilmore and R. E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14:1045–1074, 1966.

[19] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22:463–468, 1975.

[20] D. Jungnickel. *Graphs, Networks and Algorithms*. Springer, 1999.

[21] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations: Proc. of a Symp. on the Complexity of Computer Computations*, pages 85–103, 1972.

[22] H. Kellerer and U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3:59–71, 1999.

[23] H. Kellerer and U. Pferschy. Improved dynamic programming in connection with an FPTAS for the knapsack problem. *Journal of Combinatorial Optimization*, 8(1):5–11, 2004.

[24] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, illustrated edition, 2004.

[25] B. Korte and J. Vygen. *Combinatorial Optimization Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*. Springer, fourth edition, 2008.

[26] E. L. Lawler. Fast approximation algorithms for the knapsack problems. *Mathematics of Operations Research*, 4(4):339–356, 1979.

[27] E. Lin. A bibliographical survey on some well-known non-standard knapsack problems. *INFOR*, 36:274–317, 1998.

[28] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: a survey. *European Journal of Operational Research*, 141:241–252, 2002.

[29] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123:379–396, 2002.

[30] J. H. Lorie and L. J. Savage. Three problems in capital rationing. *The Journal of Business*, 28:229–239, 1955.

[31] R. Loulou and E. Michaelides. New greedy-like heuristics for the multidimensional 0-1 knapsack problem. *Operations Research*, 27:1101–1114, 1979.

[32] lp_solve. https://sourceforge.net, Accessed on August, 2008.

[33] M. J. Magazine and O. Oguz. A fully polynomial time approximation algorithm for the 0-1 knapsack problem. *European Journal of Operational Research*, 8:270–273, 1981.

[34] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. J. Wiley, 1990.

[35] G. L. Nemhauser and Z. Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15:494–505, 1969.

[36] S. Sahni. Approximate algorithms for the 0/1 knapsack problem. *J. ACM*, 22(1):115–124, 1975.

[37] S. Senju and Y. Toyoda. An approach to linear programming with 0-1 variables. *Management Science*, 15:196–207, 1968.

[38] A. Thesen. Scheduling of computer programs for optimal machine utilization. *BIT Numerical Mathematics*, 13(2):206–216, 1973.

[39] H. M. Weingartner. Capital budgeting of interrelated projects: survey and synthesis. *Management Science*, 12:485–516, 1966.

[40] H. M. Weingartner and D. N. Ness. Methods for the solution of the multidimensional 0/1 knapsack problem. *Operations Research*, 15:83–103, 1967.

[41] M-H Yang. An efficient algorithm to allocate shelf space. *European Journal of Operational Research*, 131:107–118, 2001.