UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

KNAPSACK PROBLEMS;

METHODS, MODELS AND APPLICATIONS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

In partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

KAYODE BADIRU
Norman, Oklahoma
2009

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/320340

KNAPSACK PROBLEMS;
METHODS, MODELS AND APPLICATIONS

A DISSERTATION APPROVED FOR THE
SCHOOL OF INDUSTRIAL ENGINEERING

BY

_____
Dr. Pakize Pulat - Chair

_____
Dr. Yongpei Guan - Co-chair

_____
Dr. Thomas Landers

_____
Dr. Randa Shehab

_____
Dr. Sridhar Radhakrishnan

© Copyright by KAYODE BADIRU 2009
All Rights Reserved.

Acknowledgements

I would like to thank GOD for his mercies, grace, guidance, protection, provision, and blessings upon my life. All these wouldn't have been possible without HIM.

My sincere appreciation to Dr. Simin Pulat, my advisor, for her support, encouragement, assistance, and contributions throughout this research. Also, I'm very grateful to Dr. Yongpei Guan, my co-advisor, for his time, help, and useful evaluation of the accuracy of this research. Special thanks to Dr. Tom Landers, Dr. Theodore Trafalis, Dr. Randa Shehab, and Dr. Sridhar Radhakrishnan for their time. I'm deeply honored to have them as my committee members.

My gratitude goes to my family, my wife and boys, for their continued support, anxiety, patience, and concerns. I would also like to thank Dr. Badiru, Dr. Osisanya, Mr. Adeoye, Pastor Okusanya, and families. Special thanks to Reverend Olanrewaju, Reverend Adebowale, Pastor Blessing, and my late Pastor, Pastor Okusanya, who would have greatly rejoiced with me on the completion of this research.

I dedicate this research to my wife, Mary and boys, Tobi and Wola.

Contents

Acknowledgements iv

Abstract xii

1. Introduction 1

1.1 Overview 1

1.2 Research Objectives 2

1.3 Organization of the Dissertation 3

2. Literature Review 5

2.1 Knapsack Problems 5

2.2 Tabu Search 16

2.3 Dynamic and Stochastic Knapsack Problems 20

3. Multiple Knapsack Problems with Assignment Restrictions 23

3.1 Statement of the Problem 23

3.2 Successive Knapsack Algorithm 24

3.3 Selective Successive Knapsack Algorithm 25

3.4 Largest Unutilized Capacity First Algorithm 26

3.5 The Assignment Procedures 27

3.6 Data Generation 31

4. Tabu Search 34

4.1 Properties of Tabu Search 34

4.2 Problem Definition 36

4.3 Solution Method 37

4.4 Steps of Tabu Search Procedure 38

5. Stochastic Knapsack Problems with Penalty Cost 44

5.1 Introduction 44

5.2 Notations 45

5.3 Mathematical Formulation for n-job 1-processor Case 46

5.4 n-Job 1-processor Numerical Example 48

5.5 Job Assignment Scenarios 51

5.6 Convexity of Cost Function 53

5.7 The n-job m-processor Problem 57

5.8 The n-job m-processor Problem Formulation 58

5.9 The Solution Method for n-job m-processor Problem 59

5.10 Numerical Example Data Generation 61

5.11 Numerical Example Results 62

6. Inspection Problem – an SKPPC Problem 63

6.1 Introduction 63

6.2 Problem Formulation 63

6.3 Algorithm 65

6.4 Numerical Example 67

6.5 Experimental Results 71

6.6 Further Experimentation 84

6.7 Results and Analysis 95

7. Conclusion and Further Research 97

7.1 Conclusions 97

7.2 Further Research 99

Bibliography 100

Appendix A: MATLAB Code for the Assignment Algorithms 104

Appendix B: MATLAB Code for the Tabu Search Implementation in MKAR 114

Appendix C: MATLAB Code for the SKPPC 137

Appendix D: MATLAB Code for the Inspection Problem 141

List of Tables

Table 1: The Assignment Procedures 30

Table 2: Results of unutilized capacities of all Assignment Procedures 33

Table 3: Results of Tabu Search Implementation on MKAR 43

Table 4: The two possible realizations of each item 47

Table 5: 2-job assignment results for selection of 2, 3, 4, 5, and 6 items 49

Table 6: 2-job assignment results for selection of 7, 8, 9, and 10 items 50

Table 7: Results of 1st SKPPC Problem 62

Table 8: Scenarios for 2-inspector, 4-package assignments 65

Table 9: Penalty Cost Results for Some Scenarios with different utilization penalty 68

Table 10: Total cost as a function of *n* and *m* 69

Table 11: Capacity of 20 results for 5 values of λ_1 71

Table 12: Capacity of 30 results for 5 values of λ_1 72

Table 13: Capacity of 40 results for 5 values of λ_1 73

Table 14: Capacity of 50 results for 5 values of λ_1 74

Table 15: Package type 1 probability of 0.2 results for 5 values of λ_1 76

Table 16: Package type 1 probability of 0.3 results for 5 values of λ_1 77

Table 17: Package type 1 probability of 0.4 results for 5 values of λ_1 78

Table 18: Package type 1 probability of 0.2 results for 5 values of λ_1 79

Table 19: Processing times of 16 and 1 results for 5 values of λ_1 80

Table 20: Processing times of 14 and 3 results for 5 values of λ_1 81

Table 21: Processing times of 12 and 5 results for 5 values of λ_1 82

Table 22: Processing times of 10 and 7 results for 5 values of λ_1 83

Table 23: p1=16, p2=1 values for 5 capacities 85

Table 24: p1=16, p2=3 values for 5 capacities 86

Table 25: p1=16, p2=5 values for 5 capacities 87

Table 26: p1=16, p2=7 values for 5 capacities 88

Table 27: p1=16, p2=9 values for 5 capacities 89

Table 28: p1=14, p2=1 values for 5 capacities 91

Table 29: p1=12, p2=1 values for 5 capacities 92

Table 30: p1=10, p2=1 values for 5 capacities 93

Table 31: p1=8, p2=1 values for 5 capacities 94

List of Figures

Figure 1: Tabu Search Implementation in MKAR 39

Figure 2: Typical Job Assignment Scenarios 51

Figure 3: Addition of a job to the current assignment of S_3 with n jobs 52

Figure 4: Algorithmic steps of the 1st SKPPC Problem 60

Figure 5: Expected total inspection cost versus n and m 70

Figure 6: Graph for capacity of 20 results for 5 values of λ_1 72

Figure 7: Graph for capacity of 30 results for 5 values of λ_1 73

Figure 8: Graph for capacity of 40 results for 5 values of λ_1 74

Figure 9: Graph for capacity of 50 results for 5 values of λ_1 75

Figure 10: Graph for Package type 1 probability of 0.2 results for 5 values of λ_1 76

Figure 11: Graph for Package type 1 probability of 0.3 results for 5 values of λ_1 77

Figure 12: Graph for Package type 1 probability of 0.4 results for 5 values of λ_1 78

Figure 13: Graph for Package type 1 probability of 0.5 results for 5 values of λ_1 79

Figure 14: Graph for processing times of 16 and 1 results for 5 values of λ_1 81

Figure 15: Graph for processing times of 14 and 3 results for 5 values of λ_1 82

Figure 16: Graph for processing times of 12 and 5 results for 5 values of λ_1 83

Figure 17: Graph for processing times of 10 and 7 results for 5 values of λ_1 84

Figure 18: Graphical representation of Table 21 86

Figure 19: Graphical representation of Table 22 87

Figure 20: Graphical representation of Table 23 88

Figure 21: Graphical representation of Table 24 89

Figure 22: Graphical representation of Table 25 90

Figure 23: Graphical representation of Table 26 91

Figure 24: Graphical representation of Table 27 92

Figure 25: Graphical representation of Table 28 93

Figure 26: Graphical representation of Table 29 94

Figure 27: Graph of 2-inspector problem with p_2 increased to 3 96

Abstract

Knapsack problem (KP) has broad applications in different fields such as machine scheduling, space allocation, and asset optimization. Meanwhile, it is a hard problem due to its computational complexity, but numerous solution approaches have been developed for a variety of KP. In this dissertation, an extensive literature review is first provided. Then, the research focuses on methods, models, and applications for two variations of Knapsack problem: Multiple Knapsack Problem with Assignment Restrictions (MKAR) and Stochastic Knapsack Problem with Penalty Cost (SKPPC).

A new procedure, Largest Unutilized Capacity First Algorithm (LUCF) is developed and tested on MKAR along with other assignment procedures available in the literature. It is concluded that LUCF performs very well and it returns the best initial feasible solution among all types of greedy algorithms for the solution of the MKAR. After the generation of initial feasible solutions, a tabu-search procedure is implemented to generate improved solutions. Three versions of intensification procedures are implemented within the tabu search procedure. Experimental results show significant improvement over the initial solution quality with the tabu search procedure. That is, this approach yields a high percentage of utilization for all combinations of problems, based on the initial solution provided by LUCF.

For SKPPC, for each item of the knapsack, there are several possible processing times, each with certain probability of selection. For a given knapsack capacity, a strategy is developed to assign the optimal number of items to each the knapsack. Mathematical formulations are provided for both single knapsack and *m*-knapsack cases. The objective value function for the single knapsack problem exhibits a convex

property, which leads to an optimal strategy to assign the number of items. For the *m*-knapsack case, the processing time of each item will be revealed after pre-scan operations. LUCF heuristic is combined here to obtain good solutions. This approach is finally adapted to the package security inspection problem. We discuss how one can determine the optimal number of items in each knapsack and the optimal number of operators needed for inspection with the objective of maximizing operator utilization and throughput.

Chapter 1

Introduction

1.1 Overview

The pioneering work of Dantzig [7] in the late 1950's has been followed by numerous researches in the area of Knapsack Problems (KP). These problems have been studied extensively and intensively since then (Pisinger [34]). In the most general sense, the problem deals with the assignment of a set of items into a number of knapsacks with each item having size and value associated with it. The objective is to maximize the total value of assigned items while observing the capacities of the knapsacks.

Many theoretical studies of knapsack problems have been intended and applied to the real-life problems. Many, that were mostly applications oriented, made researchers and practitioners look for better and fast solutions to cope with the vast industrial and financial management problems (Pisinger [34]).

Knapsack problems are usually sub-problems of more complex combinatorial optimization problems, and most of them require the selection of a subset of some given items resulting in the maximization of a profit sum, with the total assigned weight not exceeding the capacity of the knapsack(s). All knapsack problems are classified as being NP-hard, meaning that their optimal solutions cannot be obtained by the application of polynomial time algorithms. However, several years of research have

exposed the structural properties of these problems making them easier to solve (Pisinger [34]).

The knapsack problems have a variety of real life applications including financial modeling, production and inventory management systems, stratified sampling, design of queuing network models in manufacturing, and control of traffic overload in telecommunication systems. Other areas of applications include yield management for airlines, hotels and rental agencies, college admissions, quality adaptation and admission control for interactive multimedia systems, cargo loading, capital budgeting, cutting stock problems, and computer processing allocations in huge distributed systems.

1.2 Research Objectives

Multiple Knapsack Problem (*MKP)* generally is the assignment of items into several knapsacks. The items usually have weights, and costs associated with them, which may vary from item to item. The knapsacks may be of different capacities as well. Stochastic Knapsack Problem (*SKP*), on the other hand, assumes that the weight of the item is not known until it is placed in the knapsack. However, the weight is assumed to follow a probability distribution. The assignment of items to a knapsack generally works with the actual weights of the items already assigned and the probability distribution of the unassigned items. The objective in both cases is either to maximize capacity (or expected capacity) utilization or the most cost effective assignment. Knapsack capacities are usually never exceeded in the final assignment.

In this study, a new greedy algorithm that yields very attractive initial solution for the *MKP* is proposed. This algorithm's performance was then compared with that of

the most common assignment procedures. Multiple Knapsack Problems with Assignment Restrictions (*MKAR*), a new variant of *MKP*, is studied in regards to obtaining initial feasible solutions using the most common assignment procedures. Tabu search was later employed to improve the initial solutions generated to yield better results.

Stochastic Knapsack Problems with Penalty Cost (SKPPC) having different item types is studied to determine optimal assignment. Only two item types were studied, and each item type has a possible processing time determined by a probability of selection. The problem was extended to multiple processors for different processing times. Expected penalty cost and percentage of utilization were recorded for various problem sizes.

1.3 Organization of the Dissertation

Chapter 2 comprises the literature review on knapsack problems, its solution procedures, and some common application areas. Also included is explanation of tabu search, and dynamic and stochastic knapsack problem.

Chapter 3 contains a variant of the multiple knapsack problem - multiple knapsack problem with assignment restriction (MKAR). This was the start of the research for this dissertation. A new algorithm, *Largest Unutilized Capacity First*, LUCF was developed and tested against other known greedy procedures for assigning items to knapsacks.

Tabu Search, an efficient search method, is the discussion of Chapter 4. The tabu search procedure was applied to the initial solution generated for the MKAR. The LUCF algorithm was one of the methods used to generate an initial feasible solution.

Stochastic knapsack problem with penalty cost (SKPPC) is the topic of discussion in both Chapters 5 and 6. A study of SKPPC involving the assignment of two item types with probabilities of selection was investigated. Penalties were given for both under-utilization and over-utilization of the knapsack capacity. The extension of the problem to many processors was formulated and solved. Analyses were performed on the various variables of the problem.

Chapter 7 concludes this report with summary, conclusions and suggestions for further research.

Chapter 2

Literature Review

2.1 Knapsack Problems

The basic concept of all the families of knapsack problems involve the selection of some items, each with profit and weight values, to be packed into one or more knapsacks with capacity. The item profit p_j, weight w_j, as well as the capacity c of the knapsack are all assumed to be positive integers.

Several instances of knapsack problems, despite their worst-case complexity, may have efficient solutions via heuristic methods with acceptable computational times. The heuristics take advantage of the well defined structures inherent in these problems.

Dantzig [7] was the first to order items according to their profit-to-weight ratio, and then find a solution for the continuous 0-1 knapsack problem.

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

The ordering of the items according to this ratio can be done in *O(nlogn)* time (Dantzig [7]). The continuous 0-1 knapsack problem has its constraints on $x_j \in \{0,1,\ldots,m_j\}$ relaxed to $0 \leq x_j \leq m_j$. A greedy algorithm is then applied on the profit-to-weight ratio to assign items to knapsack starting with the largest until we reach the first item that cannot be assigned. The first unassigned item is termed the *break item b* ($b = \min\{j : \sum_{i=1}^{j} w_i > c\}$) resulting in an initial feasible solution. The optimal solution

can then be the selection of all items $j < b$ plus the residual of the knapsack capacity which can be represented by a fraction of *item b*. This procedure is utilized frequently for various types of knapsack problems.

Dynamic programming generates solutions to several knapsack problems in pseudo-polynomial time, meaning a time controlled by the number of items in a problem. Efficient algorithms have been developed by incorporating bounding tests in dynamic programming procedures.

Horowitz and Sahni's [20] solution approach for 0-1 knapsack problem in $O(\sqrt{2^n})$ worst-case time involves dividing the items into two sets. Two sets of feasible solutions are later merged after all feasible solutions of each set are enumerated. By recursively dividing the problem in two parts, makes the 0-1 knapsack problem solvable through parallel computation which runs in *O(log n log c)* where *n* and c are the number of items and the capacity of the knapsack, respectively.

Knapsack problems can also be solved using reduction algorithms (Martello and Toth [30]). Efficient ones have been developed which consist of fixing several decision variables at their optimal values before the problem is solved. This procedure decreases the decision space thereby resulting in efficient computations.

Martello and Toth [28] developed a branch-and-bound algorithm, which requires the solution of a 0-1 knapsack problem every time a lower/upper bound is found, for the multiple knapsack problems.

Heuristic algorithms like Tabu search and Genetic Algorithm have also appeared in recent times for the solution of knapsack problems. Chu et al. [6] proposed a genetic algorithm for the multidimensional knapsack problem. A heuristic based on tabu search

was presented by Glover and Kochenberger [13] whereby a flexible memory structure that integrates recency and frequency information of critical events during the solution process was employed.

The *0-1 Knapsack Problem (KP),* the root of all knapsack problems, involves the selection of a subset of *n* items into a single knapsack. The total profit of all items selected is to be maximized without the total weights exceeding the capacity of the knapsack. The general formulation of the problem follows:

$$\text{maximize } \sum_{j=1}^{n} p_j x_j \tag{2.1}$$

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

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

where p_j is the profit of each item, w_j is the weight, and x_j is 1 if item j is assigned to a knapsack or 0 otherwise.

Martello and Toth [27] proposed a new way of computing the upper bound for the 0-1 knapsack problem, and also presented a branch-and-bound algorithm for the same problem type. A bound-and-bound algorithm [28] defined as a tree-search technique that makes use of a lower bound to determine the branches to follow in the decision was later formulated for 0-1 multiple KP [28]. The term "bound-and-bound" was defined, for a maximization problem, as a tree-search technique that makes use of a lower-bound in determining the branch to investigate further in a decision tree.

Pisinger [33] presented a minimal algorithm for the 0-1 KP based on a dynamic programming approach, where the core problem is gradually extended and computational sorting and reduction of the core is minimal. A core (Balas and Zemel

[2]) is when only a small amount of the items are enumerated when there is a large probability of reaching to an optimal solution. It was shown that when the process terminates due to some bounding tests, the core processed is actually much smaller than the total number of solvable symmetrical core possible.

Hung and Fisk [21] developed a depth-first branch-and-bound algorithm for the solution of the 0-1 MKP by constructing successive higher levels of the decision tree either by assigning an object to a knapsack or by excluding that object from all knapsacks. This implies that every node generates m+1 descendent nodes, where m denotes the number of knapsacks. The essential steps of the algorithm are very much like those developed for the 0-1 KP by Ahrens and Finke [1].

The *Multiple-choice Knapsack Problem* (MCKP) is another variant of the 0-1 KP involving the selection of exactly one item j from N_i, where N_i denotes the number of item i available for each of the m items. This is formulated as follows:

$$\text{maximize} \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij} \tag{2.2}$$

$$\text{subject to} \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq c,$$

$$\sum_{j \in N_i} x_{ij} = 1, \qquad i = 1, \ldots, m.$$

$$x_{ij} \in \{0,1\}, \quad i = 1, \ldots, m, \text{ and } j \in N_i.$$

Several algorithms for MCKP have been presented over the last twenty years. Most of these algorithms start by solving linear MCKP (LMCKP) so as to obtain an upper bound. Dudzinski and Walukiewicz [9] showed that MCKP can be solved in

pseudo-polynomial time (a time controlled by the number of items) through dynamic programming. The two stages of solution of LMCKP are: a.) LP-dominated items are reduced by sorting the items in each class according to increasing weights, and delete some unpromising states by applying some dominance criteria; b.) a greedy algorithm is then used to solve the reduced LMCKP. Upper bound tests may be employed to fix several variables in each class to their optimal value after the two initial procedures mentioned before.

Balas and Zemel [2] with Fayard and Plateau [10] suggested considering the *core* which is a small subset of the items in the solution of a KP. A core can be found through partitioning procedure in $O(n)$ time, where n is the number of items. Martello and Toth [29] showed that the restricted KP defined on the core items can be solved easily for several classes of data in linear time. Pisinger [33] proposed a simple algorithm for solving LMCKP, as well as for deriving an initial feasible solution. Dynamic programming was later used from the starting initial solution to solve MCKP by adding new classes to the core as needed. This showed that to solve the MCKP to optimality, the consideration of a minimum number of classes are required.

Other 0-1 KP problem types include the Multidimensional Knapsack problem [6], the Bounded Knapsack problem (BKP) [36], the Unbounded Knapsack problem (UKP) [19], the Subset-Sum problem (SP) [41], the Multiple Knapsack problem (MKP) [35], the Bin-Packing problem (BP) [26], the Multiple-Constrained Knapsack problem [10], the Generalized Assignment problem (GAP) [5], the Quadratic Knapsack problem (QKP) [4], and the Precedence Constrained Knapsack problem (PCKP) [22].

The other variants that require mentioning are Nonlinear Knapsack problem [22], the Max-Min Knapsack problem [22], the Minimization Knapsack problem [22], the Equality Knapsack problem [22], the Strongly Correlated Knapsack problem [22], the Change-Making Knapsack problem [22], and the Collapsing Knapsack problem [22]. Others are the Parametric Knapsack problem [22], the Fractional Knapsack problem [22], the Set-Union knapsack problem [22], and the Multiperiod Knapsack problem [22].

Pisinger [35] developed and implemented an exact algorithm for large multiple knapsack problems. The MKP is defined as the assignment of some of n items into m knapsacks, where the knapsacks may be of different capacities. The aim of the problem is to maximize total profit in a way that the capacity c_i of any knapsack is not exceeded. This is formulated as follows:

$$\text{maximize} \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{2.3}$$

$$\text{subject to} \sum_{j=1}^{n} w_i x_{ij} \le c_i, \qquad i = 1,...,m.$$

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

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

x_{ij} is 1 if item j is assigned to knapsack i, or 0 otherwise. All coefficients p_j, w_j, and c_i are assumed positive integers.

The following assumptions are also essential to avoid trivial cases:

1. $\max_j \{ w_j \} \le \max_i \{ c_i \}$

2. $\min_j \{ w_j \} \leq \min_i \{ c_i \}$

3. $\sum_{i=1}^{n} w_j > \max_i \{ c_i \}$

The first assumption ensures each item is admissible into at least one knapsack or else, it may be discarded from the problem. The second assumption deals with the fact that if any item cannot fit into the smallest knapsack, the knapsack can be excluded from the problem. The last inequality assures that all items will not fit into the largest knapsack. The paper [35] is devoted to large problem situations where the ratio *n*/*m*, ratio of number of items to number of knapsacks, is very large.

The algorithm presented in the paper [35] incorporates some well know procedures to achieve its goal. The algorithm uses Martello and Toth's [28] bound-and-bound framework. A series of subset-sum problems are solved to obtain lower-bounds as well as tighten the knapsacks capacity constraints. The algorithm derives upper-bounds by incorporating a well-performing 0-1 knapsack problem through surrogate relaxation. Surrogate relaxation (by Lagrangean strategy) involves the replacement of the original objective function by a new set of constraints, the surrogate constraints. A separable dynamic programming algorithm is used for solving the subset-sum problems, and items that cannot be assigned are eliminated by efficient reduction rules which are rules for reducing a KP.

Upper-bound is derived by using surrogate relaxation on some of the side constraints. The SMKP may be formulated thus:

$$\text{maximize} \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \qquad (2.4)$$

$$\text{subject to } \sum_{i=1}^{m} \pi_i \sum_{j=1}^{n} w_i x_{ij} \le \sum_{i=1}^{m} \pi_i c_i, \qquad i = 1,...,m.$$

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

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

The best choice of multipliers of the surrogate relaxed problem, SMKP, is a positive constant *k* (where *k* is a positive number) as proved by Martello and Toth [28]. The choice of these multipliers turns the SMKP into:

$$\text{maximize } \sum_{j=1}^{n} p_j x_j^l \tag{2.5}$$

$$\text{subject to } \sum_{j=1}^{n} w_i x_j^l \le c,$$

$$x_j^l \in \{0,1\}, \qquad j = 1,...,n.$$

The introduced variables $x_j^l = \sum_{i=1}^{m} x_{ij}$ shows whether item *j* is chosen for any of the knapsack *i*, where *i* = 1to *m*, and likewise $c = \sum_{i=1}^{m} c_i$ represents the capacity of all knapsacks.

This paper [35] also utilizes the bound-and-bound algorithm of Martello and Toth [28], MTM, to derive both the lower-bounds and upper-bounds. Lower-bounds are found by solving *m* individual 0-1 knapsack problems. Upper-bounds are generated from the results of the surrogate relaxed problems.

Knapsacks are ordered in increasing order of capacities, $c_1 \le c_2 \le ... \le c_m$, and filled one after the other in that order. All assigned items are considered permanent by the branching process, and only the unassigned items are considered when lower and

upper bounds are being computed. The procedure is terminated when the gap between the lower and upper bound can no longer be tightened.

Hifi et al. [18] developed and proposed several heuristics for approximately solving the multiple-choice multidimensional knapsack problem, MMKP, which is an *NP-hard* combinatorial optimization problem. The MMKP is a more complex version of the 0-1 knapsack problem, whose high computational complexity in the formulation of an exact solution makes it unsuitable for real-time decision making applications.

The MMKP has n classes J_i of items, with each class J_i, $i = 1, \ldots, n$, consisting of r_i items. Each item j, where $j = 1, \ldots, r_i$, of class J_i has the profit value v_{ij}, non-negative, and requires resources of weight vector, $W_{ij}=(w^1_{ij}, w^2_{ij}, \ldots, w^m_{ij})$ with the component of each weight wk_{ij}, $k = 1, \ldots, m$ non-negative. A vector $C = (C^1, C^2, \ldots, C^m)$ represents the amount of available resources.

The MMKP is formulated as below:

$$\text{maximize} \sum_{i=1}^{n}\sum_{j=1}^{r_i} v_{ij} x_{ij} \tag{2.6}$$

$$\text{subject to} \sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij}^k x_{ij} \le C^k \quad k = 1,\ldots,m.$$

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

$$x_{ij} \in \{0,1\}, \qquad i = 1,\ldots,n, \qquad j = 1,\ldots,r_i.$$

The MMKP aims to pick exactly one item from each class in order for the total profit of items picked to be maximized, subject to available resource constraints.

A feasible solution exists for all $k \in \{1,...,m\}$, where $\sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leq C^k$ and only one item is picked from each class. That is, x_{ij} is 1 if item j of the ith class J_i is picked, or 0 otherwise.

Hifi et al. [18] discuss three algorithms. The first two are considered constructive and complementary solution approaches, while the third uses a *guided local search* (GLS) method.

The *GLS* algorithm by Hifi et al. [18], which is considered to be a metaheuristic, is similar to tabu search because of its memory utilization to propel the search to promising regions. It includes a penalty term in the objective function to avoid revisiting undesirable features of the previously visited solutions. The algorithm has proven to be effective in solving some hard combinatorial optimization problems. *GLS* has also been used effectively for the traveling salesman problem, quadratic assignment problem, and resource allocation. It has also been applied on vehicle routing and bin-packing problems.

The aims of the algorithm are:

a. use a greedy algorithm to start at a lower bound
b. improve the quality of the initial solution by using the *CP*
c. propel the search to the neighborhood for improvement of the solution by applying *CCP*

A pseudo-utility ratio is computed for each item by the following formula:

$u_{ij} = \frac{v_{ij}}{\langle C, W_{ij} \rangle}$, $j \in \{1,...,r_i\}$, where $\langle .,. \rangle$ is a scalar product.

The items are arranged in decreasing order of this pseudo-utility ratio, and are assigned starting with the largest, picking only one item from each class, until all classes have been covered.

The complementary procedure, CP, comprises of an ADD and a DROP phase.

The steps of *CP* are:

1. assign items using the pseudo-utility to pick the best from each class
2. *CP* terminates if the obtained solution is a feasible state, *FS*
3. for an unfeasible state, *US*, the DROP phase considers the most violated constraint in the *C*
4. the class of the largest weighted item of the most violated constraint is selected
5. the ADD phase selects another item from this class, and swap with the previous item of the most violated constraint
6. if the new state is still US, another item is swapped with the just selected one, and this continues until an FS or the smallest unfeasibility amount for the obtained solution is reached.

The complementary CP approach, *CCP*, utilizes an iterative improvement of the initial feasible solution.

The steps of *CCP* are:

1. a swapping strategy of picked items

2. a replacement stage which consists of replacing the previously assigned items with a new one selected from the same class.

2.2 Tabu Search

The tabu search method was developed by Glover [14] to solve combinatorial optimization problems. Combinatorial optimization problems, by definition, have a large discrete solution space. Tabu search imposes restrictions on the search process while rummaging around the feasible region (Glover and Laguna [15]). The search makes use of both short-term and long-term memories. The short-term memory is used to perform moves by exploring neighborhood points while long-term memory aids in the intensification of the search once an improving direction is found or in the diversification of the search to areas previously unexplored. The tabu search method can be used to guide any process that employs a set of moves for transforming one solution into another and offers an estimation of the function for measuring the attractiveness of these moves (Glover [11]).

The tabu search method can initially be viewed as a form of neighborhood search (Glover and Laguna [15]). For the neighborhood search, a current solution has an associated set of neighbors in the feasible region. The objective function is evaluated at each neighboring point and compared against the objective function value of the current point to determine the next “move”.

The tabu search procedure moves from one point to another in an effort to locate the global optimum. The procedure has the ability to escape from a local optimum by accepting a sequence of "non-improving" moves. At all stages, a tabu list is kept of moves that the procedure is not allowed to make (Pinedo [32]). The list contains a fixed number of entries. Every time a move is made in the neighborhood of the current point, the previous point is recorded at the top of the tabu list and other entries are shoved down one position while the bottom entry is removed. The size of the tabu list should not be too small to prevent cycling, but a big list of tabu moves unduly constrains the search.

For unconstrained optimization, Prabandari [37] used tabu search to find starting points for optimization techniques. Each local point, when coupled with a local optimization technique for unconstrained optimization problems, is expected to converge to a different local point.

Quadratic assignment problem (QAP) deals with the assignment of n objects to n locations in a way to minimize the total distance times flow measure between the locations (Skorin-Kapov [40]). Methods for QAP involve two phases: construction and improvement. Skorin-Kapov [40] incorporated tabu search into the improvement phase of the quadratic assignment problem to continue the search beyond local optimality.

Traveling salesman problem (TSP) is finding a complete tour that minimizes the total distance travelled by a salesman while visiting all of the n cities once, only once and returning to the starting city. TSP is a special case of QAP. The assignment problem is to ensure that the salesman visits all the cities once and terminates his

journey at the same city from where he started. Knox [25] used tabu search as a tour improvement algorithm by switching the position of points in the tour.

Facility layout is the arrangement of departments within a facility. Premkumar [38] used tabu search to find a layout better than the initial layout of a plant simulation layout (PSL) software while minimizing the cost involved in doing so.

Pinedo [32] used tabu search to reduce the number of tardy jobs on a single machine. The neighborhood of a schedule with a lower tardiness of jobs was sought through adjacent pair wise interchanges of jobs. A tabu list of jobs that were swapped recently was also kept.

Tabu search based procedure for Solving 0-1 MultiObjective Knapsack Problem, the Two Objective Case was developed by Xavier and Arnaud [42]. The paper addresses a case of MultiObjective Combinatorial Optimization (*MOCO*) Problems, the so called 0-1 MultiObjective Knapsack (0-1 MOKP).

MOCO can be formulated as below:

$$\text{maximize } z^1(x), z^2(x), \ldots, z^p(x) \qquad (2.7)$$

$$\text{subject to } x \in X .$$

X is a discrete subset of $\Re^n$, defines the decision space. $z^1(x), z^2(x), \ldots, z^p(x)$ are p objective functions.

The difficulty of *MOCO* arises due to research of all elements of the *efficient frontier*, E(P) that grows with number of objective functions. Tabu search, TS, was introduced for MOCO problems because of its efficiency in obtaining good solutions for many mono-objective combinatorial problems.

The 0-1 MOKP can be formerly formulated as below:

$$\text{maximize } \sum_{i=1}^{n} c_i^j x_i \qquad j = 1,..., p. \tag{2.8}$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \le \omega$$

$$x_i \in \{0,1\}, \qquad j = 1,..., n.$$

All coefficients, c^j_i, w_i and ω, are positive integers.

The basic steps of the algorithm developed in the paper [42] are:

1. the use of a greedy algorithm to obtain an approximation of *supported efficient solutions* SE(P)
2. tabu search is used and any potential solution *x* that is generated, is added to the set of approximate solutions if it dominates some solutions, and the solutions dominated are removed
3. a decision space reduction method is then employed

The decision space reduction method used involves the introduction of an additional constraint. Glover first introduced the bounds used by Xavier and Arnaud [42], which are

$LB = \max\left\{ s \left| \sum_{i=1}^{s} w_i \le \omega \right. \right\}$, *($w_i$) sorted in decreasing order*, and

$UB = \max\left\{ s \left| \sum_{i=1}^{s} w_i \le \omega \right. \right\}$, *($w_i$) sorted in increasing order*.

The addition of an extra constraint, $\sum_{i=1}^{n} x_i = b$, $b \in$ [LB, UB], allows for the reduction of the decision space containing both dominated feasible and infeasible solutions.

The *Tabu Search Based Procedure* (TSBP) components of Xavier and Arnaud's paper [42] were investigated by two algorithmic variations. The first starts with one initial feasible solution, and then explores one layer after the other in the solution space until a defined termination condition is satisfied. The other uses a greedy mechanism to generate an initial feasible solution. The exploration uses the information identified by the greedy algorithm to move from one layer to the other until a stopping criteria is reached.

Tabu search has also appeared in recent times for the solution of knapsack problems. A heuristic based on tabu search was presented by Glover and Kochenberger [13] whereby a flexible memory structure that integrates recency and frequency information of critical events during the solution process was employed. Glover and Lokketangen [16] developed a tabu search approach for solving zero-one mixed integer programming problems. A new approach to tabu search that provides a balance between intensification and diversification strategies was proposed by Hanafi and Freville [17].

2.3 Dynamic and Stochastic Knapsack Problems (DSKP)

Kleywegt and Papastavrou [24]'s definition of DSKP is as follows. Items, having associated reward, demand (size) for a limited resource (the knapsack) arrives according to a Poisson process in time. There is a joint distribution according to a known probability between the resource requirements and rewards which becomes known at the time of item's arrival. An item is either accepted or not. A reward is recorded for an acceptance and a penalty is incurred for a rejection. The problem can be

stopped at any time yielding a terminal value which may be due to the amount of resources remaining. The objective is to maximize the expected value (rewards minus costs) accumulated given a waiting cost within a time horizon.

A classical SKP involves the assignment of items with known sizes, or weights, into a knapsack having a fixed capacity. The objective is to maximize profit/reward. Resources that have weights and probabilities are requested and assigned to a knapsack with a fixed capacity. A typical example involves items arriving randomly over time which must either be accepted or rejected on the spot without consideration of complete information. This information includes the arrival time, the amount requested and the associated rewards derived from such operation.

The stochastic knapsack problem has been studied by Ross and Tsang [39]. They looked at a knapsack with an integer volume capable of holding different classes of objects. Objects are assumed to arrive randomly to be assigned to the knapsack, and the arrival is exponentially distributed with mean depending on the system state. They worked on finding a procedure to maximize the average revenue by either accepting or rejecting an object.

Dynamic and stochastic knapsack problem (DSKP) was the title of the paper published by Kleywegt and Papastavrou [23]. Their paper outline a scenario whereby items to be assigned to knapsack arrive according to Poisson process in time. Associated with each item, is its reward, size, and a limited resource. The item's reward is received if it is accepted and a penalty is paid if rejected. The resource requirement and reward of an item are jointly distributed according to a known probability distribution. These become known when the item arrives.

Papastavrou et al. [31] included deadlines in their study of the DSKP. The problem definition is the same as previously described with the addition of fixed time horizon. They determined the optimal policy for the knapsack within the time allowed in order to maximize the expected accumulated reward.

Kleywegt and Papastavrou [24] improved on their previous work by having items with random sizes. Their objective was to determine the maximum expected value (rewards minus costs) accumulated. A reward is received if an item is accepted, and a penalty is incurred if rejected. The resource requirements and rewards are known at the time of the demand's arrival, but unknown before then. They showed that the DSKP has an optimal assignment that includes both an easily computed threshold acceptance rule and an optimal stopping one.

The stochastic knapsack problem (SKP) to be studied involves items with possible processing times. The processing times are unknown but have probabilities of being selected. The objective of the problem would be to minimize the expected penalty cost of all assignments. Two versions of the Stochastic Knapsack Problem with Penalty Cost, *SKPPC*, would be investigated; the one processor and many processors case. Both the expected penalty cost and percentage of utilization would be recorded.

Chapter 3

Multiple Knapsack Problems with Assignment Restrictions (MKAR)

3.1 Statement of the Problem

The *Multiple Knapsack Problem with Assignment Restrictions* (MKAR) is a variant of the well-studied *Multiple Knapsack Problem* (MKP), which is a generalization of single *Knapsack Problem* (KP). The MKAR deals with items that are constrained to particular knapsacks. The problem to be solved is to maximize assigned weights for each knapsack, with due consideration to the assignment restrictions. The formal representation of this kind of knapsack problem is described as follows:

$$\text{maximize} \sum_{i \in M} \sum_{j \in B_i} w_j x_{ij} \tag{3.1}$$

$$\text{subject to} \sum_{j \in B_i} w_j x_{ij} \leq c_i, \qquad i \in M$$

$$\sum_{j \in A_j} x_{ij} \leq 1, \qquad j \in N$$

$$x_{ij} \in \{0,1\}, \qquad i \in A_j \quad j \in N,$$

where, the variable x_{ij} indicates whether an item j is assigned to a knapsack i.

The MKAR can be described as follows:

N The set of items to be assigned, $N = \{1, \ldots, n\}$

M The set of knapsacks to be filled, $M = \{1, \ldots, m\}$

w_j The weight of item j

p_j The profit of item j

c_i The capacity of knapsack i

A_j The set of knapsacks that can hold item j, A_j *is* a subset of M

B_i The set of items that can be assigned to knapsack i, B_i *is* a subset of N

A feasible assignment is one in which:

- Each item is assigned to at most one knapsack,
- Assignment restrictions are satisfied,
- Total weight of items assigned to a knapsack does not exceed its capacity.

The following assumptions can be made:

$w_j, p_j > 0$ and integers for all j in N

$c_i > 0$ and integer for all i in M

$\min_j \{ w_j \} \leq \min_i \{ c_i \}$

$\max_j \{ w_j \} \leq \max_i \{ c_i \}$

$$\sum_{i=1}^{n} w_j > \max_i \{ c_i \}$$

Dawande et al. [8] started the pioneering work in this area of knapsack problems and developed two major algorithms, which are successive knapsack, and selective successive knapsack algorithms.

3.2 Successive Knapsack Algorithm (SK)

This is the same as maximizing assigned weight under assignment restrictions. This simple algorithm assign items to knapsacks one after the other. One knapsack is completely filled before going to the next. The procedure of the algorithm is as follows:

1. Initialize $S = N$, $Weight_i = 0$

2. For each knapsack i

   2.1 Solve a single knapsack problem for each knapsack i with item set $S \cap B_i$

   2.2 Let S_i be the set of items packed with total $Weight_i$

   2.3 Remove S_i from S

3.3 Selective Successive Knapsack (SSK) Algorithm

This is the other algorithm presented in the paper by Dawande et al. [8] which is bi-criteria. It involves maximizing assigned weight and minimizing total unused capacity. The steps of the algorithm follow:

Initialize $S = N$, $R = M$, $Weight_i = 0$, $AW = 0$ (total assigned weight)

(1) For all $i \in R$, calculate $Weight_i$ and $Waste_i$ by solving a single knapsack problem for knapsack i with set $S \cap B_i$. $Weight_i$ is the total weight of assigned items in knapsack i, $Waste_i$ is the unutilized space of the knapsack i.

(2) Pick the knapsack with minimum ratio of $Waste_i$ / $Weight_i$, say knapsack k.

(3) Pack items into knapsack k to obtain $Weight_k$, add $Weight_k$ to AW.

(4) If $AW \geq$ T/3, then terminate the algorithm.

(5) Otherwise,

   (5.1) Remove assigned items from S and knapsack k from R.

   (5.2) If R is nonempty, go to Step (1).

   (5.3) If R is empty, terminate the algorithm.

Research was conducted which involves proposing a similar algorithm to the SSK to solve this kind of knapsack problem (Dawande et al. [8]). It was realized that the SSK generates solution using the procedure of subset-sum problems, hence the decision to investigate to see if there can be an improvement on this procedure by using a different approach. A procedure called the *largest unutilized capacity first* (LUCF) algorithm was developed. It is a greedy algorithm that arranges both the items and knapsacks in non-decreasing order of their values, and assigns the next item to the knapsack with the largest unutilized capacity. The items are arranged with $w_1 \geq w_2 \geq ... \geq w_m$, while the knapsacks are ordered such that $c_1 \geq c_2 \geq ... \geq c_n$. It was believed that, a better approach to this problem will be the selection of knapsack based upon unutilized capacity, which means selecting the knapsack with the largest available space first. The idea comes from scheduling theories whereby largest processing time first is used for allocating jobs on parallel machines to minimize completion time. Such scheduling results in load balancing amongst the machines. In this case, knapsack capacities would be balanced thereby maximizing the weight assignment of items in the various knapsacks.

3.4 Largest Unutilized Capacity First Algorithm (LUCF)

The steps of the LUCF, the algorithm developed, are as follows:

Initialize $S = N$, $R = M$, $Weight_i = 0$, $Space_i = 0$

(1) For each item j

Pick the knapsack i with $Space_i = \max_i (c_i - Weight_i)$ in A_j

(2) If $w_j > Space_i$, remove item j from S

(3) Otherwise assign item j to knapsack *i*

(3.1) $Weight_i = Weight_i + w_j$

(3.2) Remove item *j* from *S*

(4) If $\min_j \{w_j\}$ in $A_j > Space_i$, remove knapsack *i* from *R*

(5) If *R* is nonempty, go to Step (1)

(6) If *R* is empty, terminate the algorithm.

Ten different ways of assigning items to knapsacks were modeled and studied. The same data set was used for all the ten procedures. The performance of all were recorded and compared against the proposed LUCF algorithm.

3.5 The Assignment Procedures

The ten assignment procedures can be divided into two major groups; those with smallest items assigned first and ones with largest items first.

Procedure 1 had both the items and the knapsacks arranged in increasing order, that is, smallest items with smallest knapsacks. The items were then assigned to the knapsacks simultaneously. This means that the next item was assigned to the next knapsack in line. This continues until no knapsack has enough space to accept the next item.

Procedure 2 also had both the items and the knapsacks arranged in increasing order. However, unlike procedure 1, the next knapsack was filled completely until the

next item cannot be assigned. This procedure was continued until all knapsacks have been assigned the most items.

Procedure 3 is similar to procedure 1 in the aspect of assignment, but the items and knapsacks are initially arranged in opposite order, The items are arranged in increasing order, smallest items first, while the knapsacks are in non-decreasing order, largest knapsacks first. The items were then assigned into the knapsacks.

Procedure 4 had the same item and knapsack arrangement as procedure 3, but similar assignment as procedure 2. The next knapsack is filled completely before processing to the next.

Procedure 5 had the items arranged in decreasing order and the knapsacks arranged in increasing order. This was largest items-smallest knapsacks setup. Items were then assigned simultaneously into the knapsacks.

Procedure 6 had the items arranged in non-decreasing order, and the knapsacks arranged in increasing order. The assignment was carried out with each knapsack completely filled before the next.

Procedure 7 was one of the three setups that had both the items and knapsacks arranged in non-decreasing order. It was largest items, largest knapsacks procedure. The items were then assigned simultaneously into the knapsacks until no item could be admissible by any knapsack.

Procedure 8 shared the same items and knapsack arrangement with procedure 7. However, knapsacks were completely filled one after the other during assignment. This continued until the next item couldn't fit into the knapsack with the largest unused

space. The procedure was stopped at this point since the remaining knapsacks would have smaller unused space.

Procedures 9 and 10 shared the same assignment protocol. Largest unutilized capacity knapsack was selected next during the process. This means that the next item would be assigned to the knapsack that has the biggest space. Procedure 9 had the items arranged in increasing order while procedure 10 was the opposite with the items arranged in non-decreasing order. Items were then assigned one after the other until there was no space for the next one in any of the knapsacks. Procedure 10 is the largest unutilized capacity first (LUCF) algorithm.

Items sizes of 25, 50, 100, and 200 were generated and assigned to knapsacks of capacities 2, 3, 4, and 5. The data generation steps were taken from the book by Martello and Toth [30].

The ten procedures studied are:

Table 1: The Assignment Procedures

PROCEDURE 1	Assign next smallest item to the Next Smallest knapsack
PROCEDURE 2	Fill next smallest knapsack with the Next smallest items
PROCEDURE 3	Assign next smallest item to the Next biggest knapsack
PROCEDURE 4	Fill next biggest knapsack with the Next smallest items
PROCEDURE 5	Assign next biggest item to the Next smallest knapsack
PROCEDURE 6	Fill next smallest knapsack with the Next biggest items
PROCEDURE 7	Assign next biggest item to the Next biggest knapsack
PROCEDURE 8	Fill next biggest knapsack with the Next biggest items
PROCEDURE 9	Assign next smallest item to the Knapsack with the biggest space
PROCEDURE 10 **(LUCF)**	Assign next biggest item to the Knapsack with the biggest space

3.6 Data Generation

Uncorrelated items were generated with w_j uniformly random in [10, 100], and capacities having c_i uniformly random in $\left[0.4\sum_{j=1}^{n} w_j / m, 0.6\sum_{j=1}^{n} w_j / m\right]$ for i = 1, m-1. The capacity of the mth knapsack was set to $c_m = \left[0.5\sum_{j=1}^{n} w_j - \sum_{i=1}^{m-1} c_j\right]$.

The following conditions must be satisfied for all formulations:

(1) w_j, p_j > 0 and integers for all j in N

(2) c_i >0 and integer for all i in M

(3) $\min_j \{ w_j \} \leq \min_i \{ c_i \}$

(4) $\max_j \{ w_j \} \leq \max_i \{ c_i \}$

(5) $\sum_{i=1}^{n} w_j > \max_i \{ c_i \}$

The items generated were sorted in ascending order for some procedures and in descending order for other procedures. The same was done for the knapsack capacities generated.

MATLAB and EXCEL were the computation platforms. All codes for both the generation of data sets and execution of knapsack/item assignments were performed in MATLAB. An EXCEL table was used to compare the results generated.

Twenty runs of each knapsack/item combination were executed, and the same data set was used for all the procedures at all times. The minimum, maximum, and the average of the unutilized capacities were then recorded for each procedure for the twenty replications.

All the procedures were able to fully utilize the knapsack capacity at least 50% of the time. Procedures 5, 6, 7, 8, and 10 all returned maximum utilization in at least one replication. The best performers, in all the three categories of data recorded, are procedures 6, 8, and 10.

The proposed *LUCF* (procedure 10) procedure performed very well amongst all studied procedures, and returned best initial solutions about 70% of the time. The table on the next page shows the performance comparisons of the ten procedures.

The next step, in the solution of the problem, was the design of an improvement method. Tabu search was implemented in the improvement stage of the initial solution generated by the procedures.

Table 2: Results of the unutilized capacities of all Assignment Procedures

m	n	1			2			3			4			5		
		min	max	ave	min	max	ave	min	max	ave	min	max	ave	min	max	ave
2	25	0.000	0.219	0.091	0.003	0.167	0.084	0.006	0.206	0.093	0.006	0.220	0.102	0.000	0.045	0.009
	50	0.000	0.098	0.052	0.002	0.096	0.041	0.000	0.095	0.044	0.005	0.118	0.042	0.000	0.015	0.003
	100	0.000	0.052	0.027	0.002	0.042	0.021	0.003	0.054	0.027	0.000	0.059	0.025	0.000	0.010	0.001
	200	0.001	0.025	0.011	0.001	0.023	0.011	0.000	0.027	0.012	0.000	0.028	0.011	0.000	0.003	0.000
3	25	0.016	0.289	0.156	0.000	0.268	0.115	0.004	0.280	0.165	0.000	0.331	0.126	0.000	0.243	0.032
	50	0.000	0.172	0.069	0.000	0.130	0.062	0.002	0.148	0.064	0.000	0.170	0.063	0.000	0.083	0.009
	100	0.000	0.079	0.038	0.000	0.069	0.028	0.001	0.085	0.039	0.001	0.103	0.037	0.000	0.059	0.003
	200	0.001	0.040	0.017	0.000	0.030	0.014	0.001	0.040	0.020	0.000	0.044	0.015	0.000	0.030	0.002
4	25	0.000	0.410	0.133	0.000	0.321	0.137	0.000	0.391	0.187	0.000	0.492	0.167	0.000	0.383	0.054
	50	0.000	0.218	0.102	0.000	0.172	0.076	0.000	0.217	0.094	0.003	0.243	0.084	0.000	0.174	0.019
	100	0.002	0.120	0.055	0.000	0.090	0.038	0.000	0.101	0.051	0.000	0.133	0.039	0.000	0.068	0.005
	200	0.001	0.054	0.027	0.001	0.045	0.018	0.002	0.049	0.024	0.000	0.056	0.021	0.000	0.031	0.003
5	25	0.012	0.367	0.195	0.000	0.373	0.179	0.000	0.496	0.253	0.000	0.547	0.203	0.000	0.457	0.118
	50	0.000	0.263	0.126	0.000	0.212	0.092	0.000	0.254	0.110	0.004	0.306	0.113	0.000	0.255	0.043
	100	0.000	0.158	0.064	0.000	0.101	0.044	0.000	0.133	0.062	0.000	0.141	0.055	0.000	0.101	0.016
	200	0.000	0.069	0.029	0.000	0.051	0.022	0.000	0.067	0.035	0.000	0.075	0.022	0.000	0.063	0.007

m	n	6			7			8			9			10		
		min	max	ave	min	max	ave	min	max	ave	min	max	ave	min	max	ave
2	25	0.000	0.037	0.010	0.000	0.176	0.022	0.000	0.053	0.012	0.000	0.203	0.106	0.000	0.037	0.007
	50	0.000	0.012	0.003	0.000	0.196	0.063	0.000	0.012	0.003	0.000	0.123	0.055	0.000	0.016	0.003
	100	0.000	0.010	0.001	0.000	0.272	0.086	0.000	0.006	0.001	0.000	0.058	0.026	0.000	0.008	0.001
	200	0.000	0.004	0.000	0.000	0.304	0.086	0.000	0.002	0.000	0.000	0.025	0.011	0.000	0.002	0.000
3	25	0.000	0.074	0.018	0.000	0.106	0.019	0.000	0.099	0.014	0.003	0.391	0.162	0.000	0.090	0.019
	50	0.000	0.029	0.006	0.000	0.191	0.036	0.000	0.031	0.005	0.000	0.180	0.085	0.000	0.028	0.005
	100	0.000	0.009	0.002	0.000	0.328	0.085	0.000	0.008	0.001	0.000	0.099	0.042	0.000	0.009	0.001
	200	0.000	0.006	0.001	0.000	0.322	0.096	0.000	0.005	0.001	0.000	0.043	0.019	0.000	0.007	0.001
4	25	0.000	0.100	0.018	0.000	0.245	0.025	0.000	0.115	0.029	0.004	0.494	0.211	0.000	0.127	0.022
	50	0.000	0.032	0.008	0.000	0.252	0.036	0.000	0.040	0.007	0.000	0.255	0.112	0.000	0.037	0.005
	100	0.000	0.014	0.002	0.000	0.415	0.069	0.000	0.013	0.002	0.002	0.171	0.054	0.000	0.016	0.002
	200	0.000	0.007	0.001	0.000	0.476	0.124	0.000	0.006	0.000	0.000	0.079	0.027	0.000	0.009	0.000
5	25	0.000	0.147	0.032	0.000	0.259	0.040	0.000	0.163	0.033	0.000	0.617	0.265	0.000	0.163	0.034
	50	0.000	0.053	0.010	0.000	0.202	0.020	0.000	0.070	0.010	0.000	0.342	0.133	0.000	0.064	0.010
	100	0.000	0.025	0.003	0.000	0.421	0.068	0.000	0.025	0.003	0.000	0.214	0.069	0.000	0.024	0.003
	200	0.000	0.009	0.001	0.000	0.363	0.083	0.000	0.010	0.001	0.000	0.076	0.033	0.000	0.009	0.000

Chapter 4

Tabu Search

4.1 Properties of Tabu Search

The tabu search method can initially be viewed as a form of neighborhood search (Glover and Laguna [15]). For the neighborhood search, a current solution has an associated set of neighbors in the feasible region. The objective function is evaluated at each neighboring point and compared against the objective function value of the current point to determine the next "move".

The following steps explain the neighborhood search method adapted from Glover and Laguna [15].

Step 1: Initialization

1.1 Select a starting point x^{now} in the feasible space.

1.2 Record the current "best" solution. If x^{now} is better than x^{best}, set $x^{best} = x^{now}$, else x^{best} remains.

Step 2: Decision and Termination

2.1 Choose a solution, x^{next}, from the neighborhood points of x^{now}.

2.2 Terminate if: (i) x^{next} can not be found by applying the decision criteria or

(ii) when a termination criterion is met.

Step 3: Update

Reset $x^{now} = x^{next}$, and perform Step 1(b). Return to Step 2.

The tabu search method uses the above neighborhood search strategies and builds upon the set of criteria to be employed to move from one point to another. While doing so, it employs the use of a tabu list, short-term, and long-term memory structures. Several unique terms and definitions are used in the tabu search method.

Tabu size is the number of moves in the tabu list. **Tabu list** is the set of moves that are not permitted by the search at any particular moment. The **number of restarts** is the maximum number of times the tabu search procedure is run before a particular search is terminated. Restarts diversify the search to other areas on the surface of the measured sample in hopes of obtaining an improved solution. The **number of iterations** equals the number of moves allowed within each restart. Iteration is a move from one sample point to another on the measurement surface. This could be a move from a good solution to a bad solution because the heuristic allows such moves in order to escape from local optimality. The number of iterations is always a function of the size of the sample to be measured. The **number of destroyed iterations** is the number of non-improving moves allowed within each restart. Non-improving moves, or bad moves, are moves from a current solution to a solution with an objective function value worse than the current solution. This enables the search to escape from a region of local optimal solution to an immediate neighboring region in search of a better solution. **Short-term memory** stores the best solution for each restart. The concept is that the global optimal solution should be contained in the set of good solutions. The solution for each set of iterations is recorded and the best solution picked out of the recorded solutions. **Long-term memory** stores the number of times each point has been sampled. **Intensification** is the process of focusing the search in areas where previous

best solutions were obtained. This strategy assumes that the global optimum will be in this neighborhood. **Diversification** is the process of focusing the search in unexplored areas by avoiding previously visited locations.

4.2 Problem Definition

Tabu search, a very reliable and promising search procedure, is to be applied to the *Multiple Knapsack Problem with Assignment Restrictions* (MKAR). The MKAR deals with items that are constrained to particular knapsacks.

The formal representation of this kind of knapsack problem follows:

$$\text{maximize} \sum_{i \in M} \sum_{j \in B_i} w_j x_{ij} \tag{4.1}$$

$$\text{subject to} \sum_{j \in B_i} w_j x_{ij} \le c_i, \qquad i \in M$$

$$\sum_{j \in A_j} x_{ij} \le 1, \qquad j \in N$$

$$x_{ij} \in \{0,1\}, \qquad i \in A_j \quad j \in N,$$

where the variable x_{ij} indicates whether an item j is assigned to a knapsack i.

The notations of MKAR can be described as follows:

N The set of items to be assigned, $N = \{1, \ldots, n\}$

M The set of knapsacks to be filled, $M = \{1, \ldots, m\}$

w_j The weight of item j

p_j The profit of item j

c_i The capacity of knapsack i

A_j The set of knapsacks that can hold item j, A_j a subset of M

B_i The set of items that can be assigned to knapsack i, B_i a subset of N

The utilization of the features of tabu search is expected to bring good results in reasonable time. The tabu size, tabu list, number of bad moves allowed, short and long-term memories, intensification, and diversification would be the most used for finding the solution of the MKAR. The tabu size and list would be determined by the problem sizes. The number of bad moves allowed would be set to yield good results. The short-term memory would be used to store the result of iterations, while the long-term memory would store the best overall result. The intensification step would be in three steps. The first step would be individual optimization of each knapsack assignment, while the second step is the employment of problem set reduction by fixing some items in the solution set. Third step would be the pair-wise exchange of items between knapsacks. All these are implemented after an initial feasible solution has been obtained. Diversification to other regions of the solution space would be carried out by using the solutions obtained by the various assignment methods as a starting point before the steps of the intensification procedures. The unattractive solutions, i.e. bad moves, during iterations would be allowed to see if this propels the procedure into other regions. The algorithm stops after a fixed number of iterations, and maybe by the use of a stopping termination criterion.

4.3 Solution Method

The solution would be obtained by the following major steps:

1. Implement the various assignment methods to obtain a series of feasible starting solutions.

2. Initialize tabu size (TS), tabu list (TL), number of bad moves allowed, (BM), short-term memory (SM), and the long-term memory (LM).
3. Pick the procedure with smallest unutilized capacity.
4. Perform intensification to improve on the solution obtain.
5. Perform diversification to improve solution.
6. Terminate algorithm if best assignment is obtained or after all solutions methods have been investigated.

4.4 Steps of Tabu Search Procedure

The major steps of the solution method involve 5 procedures. These procedures are the main procedure, 3 intensification procedures and a diversification procedure.

The main procedure (MP) involves obtaining initial feasible solutions from all the procedures. The procedure with the maximum utilization was then selected for the next stage, the first intensification procedure.

The first intensification procedure (IP1) consists of trying to maximize the overall capacity utilization by solving single knapsack problem for each knapsack. The result obtained was then passed over to the second intensification procedure, IP2.

Further improvement was the aim of IP2, the second intensification procedure. This involves reducing the problem size by making some items to be included in the assignment solution. That is, some items are fixed, always selected, in any assignment.

Pair-wise exchange of items between knapsacks was the purpose of IP3, the third intensification procedure. Items are exchanged between two knapsacks to see if further improvement could be made on the solution.

The last, but not the least, was the diversification procedure, DP, which is an integral part of tabu search. The next best solution, in terms of capacity utilization, amongst the remaining procedures acts as a starting solution for IP1, and the whole steps with IP2, and IP3 were repeated. This was done for all the procedures until all have been utilized.

The procedures below were implemented in MATLAB, to perform the steps outlined in solving the MKAR.

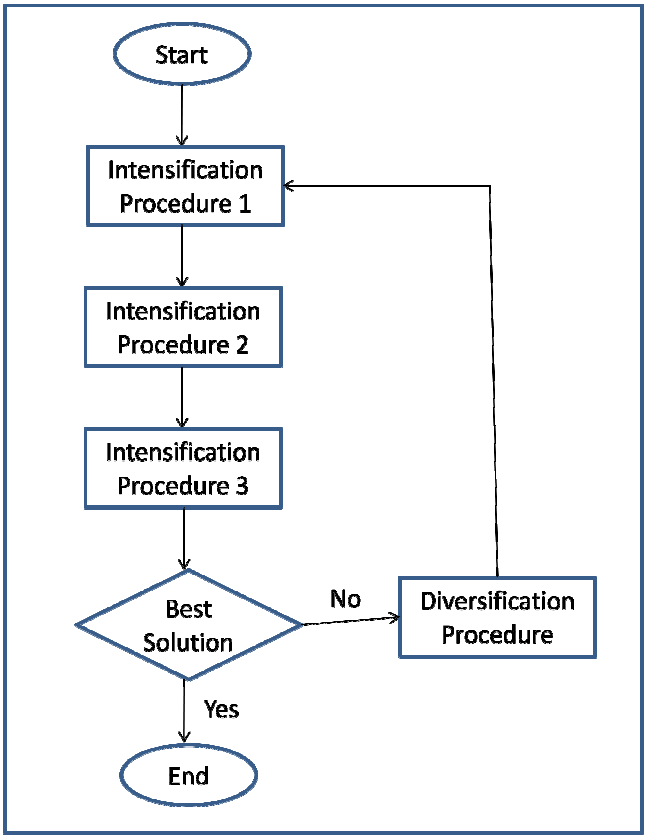


Figure 1: Tabu Search Implementation in MKAR.

4.4.1 Main Procedure (MP)

1. Solve the *MKAR* using ALL PROCEDURES
2. Save the solutions of all procedures in Mresult
3. Pick the procedure with the best solution say *Zbest* and *Xbest*
4. Make $Z^* = Zbest$, and $X^* = Xbest$
5. Remove *Mbest* from *Mresult*
6. Initialize TL = 4 and BM = 0
7. CALL IP1

4.4.2 Intensification Procedure 1 (IP1)

1. Initialize *Z, X, S=N, R=M*
2. Update *TL*
3. Pick the knapsack, say knapsack *k*, with the largest unutilized capacity
4. Solve *0-1 KP* on knapsack *k* to yield *Z* and *X*
5. If $Z > Z^*$, replace Z^* with Z, and X^* with *X*

   Else If $Z \leq Z^*$, *BM = BM +1*
6. Remove assigned items from *S* and knapsack *k* from *R*
7. If $R = \{0\}$ and BM < 4, go to step 2
8. CALL IP2

4.4.3 Intensification Procedure 2 (IP2)

1. Initialize *Z*, *X*, *S=N*, *R=M*, *BM* = 0, *F*
2. Update *TL*
3. Pick the knapsack, say knapsack *k*, with the largest unutilized capacity
4. Fix F items in knapsack *k*
5. Solve the *MKAR* using the *LUCF* algorithm on the reduced problem to yield *Z* and *X*
6. If $Z > Z^*$, replace Z^* with Z, and X^* with *X*, $F = F + 1$

   Remove assigned items from *S* and knapsack *k* from R, go to step 2
7. *If* $Z \leq Z^*$, $BM = BM + 1$, $F = F + 1$,

   If $BM = 4$, Remove assigned items from *S* and knapsack *k* from R go to step 2
8. CALL IP3

4.4.4 Intensification Procedure 3 (IP3)

1. Initialize *Z*, *X*, *S=N*, *R=M*, *BM* = 0
2. Update *TL*
3. Pick two adjacent knapsacks, say knapsacks *k1* and *k2*
4. Exchange items between knapsacks *k1* and *k2*
5. Solve *0-1 KP* on knapsack *k1* and knapsack *k2* to yield *Z* and *X*
6. If $Z > Z^*$, replace Z^* with Z, and X^* with *X*

   Else if $Z < Z^*$, $B = B + 1$
7. *If* $BM < 4$, go to step 2
8. CALL DP

4.4.5 Diversification Procedure (DP)

1. Pick the next best procedure from Mbest
2. Remove this procedure from Mbest
3. CALL IP1
4. If $Mresult \notin \{0\}$, go to step 1
5. Terminate the algorithm, and record Z^* and X^* as the best solution obtained

The performance of the procedure was measured by the percentage utilization which is defined as the percentage ratio of the total utilization by the total knapsack capacities.

$$\%Utilization = \frac{sum(utilizations)}{sum(capacities)}.$$

This was found to be more than 99% from the 16 problems solved. Table 3 contains the percentage utilization for the items/knapsack combinations generated and analyzed.

The table has data for initial solution, final solution, and the percentage increase. The initial solution was the best overall solution selected from all the initial feasible results of the ten procedures after the execution of the main procedure. The final solution is the best result obtained from all the steps of both the intensification and the diversification procedures.

The three levels of intensifications were employed to obtain the best solution possible. No comparison could be carried out to check the performance of the intensification procedures against each other because they have different starting points. It was also observed that the there was no consistency in regards to solution

improvement from one intensification procedure to another. There was little or no improvement in some cases. IP1, the first intensification procedure, seems to give the best solution improvement in most cases in terms of capacity utilization increase.

The item sizes of 25, 50, 100, and 200 were generated, and assigned into 2, 3, 4, and 5 knapsacks. The tabu search was terminated at any point a full utilization is obtained. This occurred three times in all the sixteen problems solved. Maximum utilizations were obtained six times after the initial solution were improved upon.

Table 3: Results of Tabu Search Implementation on MKAR.

# of Knapsack m	# of Items n	initial	Solution final	%increase
2	25	98.87	100.00	1.14
	50	97.39	100.00	2.68
	100	99.71	100.00	0.29
	200	100.00	100.00	0.00
3	25	98.31	99.48	1.19
	50	100.00	100.00	0.00
	100	99.96	100.00	0.04
	200	97.47	99.55	2.14
4	25	98.18	99.39	1.23
	50	100.00	100.00	0.00
	100	92.39	99.96	8.20
	200	99.98	100.00	0.02
5	25	97.72	99.19	1.50
	50	97.96	100.00	2.08
	100	99.88	99.97	0.09
	200	99.84	100.00	0.16

Chapter 5

Stochastic Knapsack Problems with Penalty Cost (SKPPC)

5.1 Introduction

In this chapter, we consider the stochastic knapsack problem with penalty cost. More specifically, we focus on the case where there is only one item type. The processing time for each item of the specified type is unknown. We assume that the processing time can take one of two possible values with probabilities associated with each value. The goal is to assign the items with unknown processing times into knapsacks in a way to minimize expected under-utilization of the knapsacks.

The first problem we studied involves a single knapsack. This study was then extended to multiple knapsacks. Although the problems bear similarity to problems discussed previously on DSKP, these have penalty cost associated with both under-utilization and over-utilization of resources. The probabilities associated with item types are known prior to the commencement of execution of any problem. This property makes it different from all DSKP, studied to date, which consider mostly dynamic probabilities. The objective is to minimize the total expected penalty cost. There is penalty cost for under-utilization, as well as over-utilization of resources. This means that the expected total cost value always has a penalty function for both cases of assignments. All item assignment combinations are generated, these are called *scenarios*, and the expected cost value calculated for each scenario.

The steps of the solution procedure, of a one-knapsack two-item-type problem would be:

1. compute all possible item assignments,
2. estimate the objective values among all scenarios, and
3. increase number of item type and repeat step 1

The motivation of our research involves security inspection of packages at the airports. Packages are categorized into two groups: high-risk and low-risk. The high-risk packages require more inspection time. The objective is to minimize total expected cost of packages inspected, and a penalty cost is incurred if the given time is not fully utilized or over-utilized.

5.2 Notations

The problem can be described either in terms of scheduling jobs or item assignment. Here, C which normally represents the knapsack capacity can also mean the available machine time in case of job processing, or resources available to processors. $c_{\max}$ would then mean the processing time of the last job as compared to the maximum assigned weight in case of knapsack assignments. Without loss of generality, we can present the problem using the following notations similar to job scheduling in a production planning environment.

First, we assume the processing time is t_1 for one realization of the processing time of each job and t_2 is the value for the other realization of the processing time of the job. The expected processing time is defined as E[T].

Objective function = minimize expected total penalty cost.

n = number of jobs

p_j = processing time of job j

α_j = probability associated with selection of job j

λ_1 = penalty for each unit time for under-utilization

λ_2 = penalty for each unit time for over-utilization

ϕ = set of all possible scenarios

C = total available machine time/resources

c_{max} = completion time of the last assigned job

5.3 Mathematical Formulation for *n*-job-*1*-processor Case

The objective is to minimize expected total penalty cost of processing a set of items. Items were assigned by selecting the number of items that generates the minimum penalty cost. Since each item has two possible scenarios, for a set of n items, there are 2^n possible scenarios.

The mathematical formulation can be described as follows:

$$\text{Minimize } \sum_{k\in\phi} S_k[\lambda_1 \max(C - c^k_{\max},0) + \lambda_2 \max(c^k_{\max} - C,0)] \quad (5.1)$$

S_k = the probability that the k^{th} scenario will happen

$c^k_{\max}$ = completion time of last job under scenario k

$c^k_{\max} = \sum_j p^k_j$ for all $k \in \phi$ where p^k_j = the processing time for job j in scenario k.

n = number of items waiting for service

For the case that all items are the same type, that is, $p_j = t_1$ with probability α and $p_j = t_2$ with probability1-α. Then to assign n jobs, we will get n+1scenarios.

For an n-job problem, this problem can be written as:

$$f(n) = \sum_{k=0}^{n} \binom{n}{k} \alpha^k (1-\alpha)^{n-k} \{\lambda_1 \max(C - c_{\max}^k, 0) + \lambda_2 \max(c_{\max}^k - C, 0)\} \quad (5.2)$$

Where,

$$c_{\max}^k = t_1 k + t_2 (n-k)$$

For instance, in the following, we list a two-item case.

Table 4: The two possible realizations of each item

	Processing time for the first item	Processing time for the second item	Probability
Scenario 1	t_1	t_1	α^2
Scenario 2	t_1	t_2	2 α (1- α)
Scenario 3	t_2	t_2	(1- α)^2

The 2-job type can be represented mathematical as below:

$$f(2) = \sum_{k=0}^{2} \binom{2}{k} \alpha^k (1-\alpha)^{2-k} \{\lambda_1 \max(C - c_{\max}^k, 0) + \lambda_2 \max(c_{\max}^k - C, 0)\} \quad (5.3)$$

If the completion time of all scenarios is less than C, then

$$f(2) = \sum_{k=0}^{2} \binom{2}{k} \alpha^k (1-\alpha)^{2-k} \{\lambda_1 (C - c_{\max}^k)\} \tag{5.4}$$

If the completion time of all scenarios is greater than *C*, then

$$f(2) = \sum_{k=0}^{2} \binom{2}{k} \alpha^k (1-\alpha)^{2-k} \{\lambda_2 (c_{\max}^k - C)\} \tag{5.5}$$

5.4 *n*-job-*1*-processor Numerical Example

Each job has two possible processing times of 16 and 1, each with the probabilities 0.2 and 0.8 respectively. The results are shown in the obtained table.

Tables 5 and 6 show the worksheet derived for this type of problem. It was noted, from the results, that the values obtained started to increase after an initial descent. This makes us conclude that the results could have a convex structure after this behavior was repeatedly obvious for some other assignment scenarios as well.

Table 5: 2-job assignment results for selection of 2, 3, 4, 5 and 6 items

	Item 1	Item 2		Knapsack			λ1	λ2		
Weight	16	1		20			1	1		
Probability	0.2	0.8								
Assign	Item 1	Item 2	Total	Space	Distributn	C-Cmax	Cmax-C	Probability	Summation	
2 items	2	0	32	-12	1		12	0.04	0.48	
	1	1	17	3	2	3		0.32	0.96	
	0	2	2	18	1	18		0.64	11.52	
								1	**12.96**	
3 items	3	0	48	-28	1		28	0.008	0.224	
	2	1	33	-13	3		13	0.096	1.248	
	1	2	18	2	3	2		0.384	0.768	
	0	3	3	17	1	17		0.512	8.704	
								1	**10.944**	
4 items	4	0	64	-44	1		44	0.0016	0.0704	
	3	1	49	-29	4		29	0.0256	0.7424	
	2	2	34	-14	6		14	0.1536	2.1504	
	1	3	19	1	4	1		0.4096	0.4096	
	0	4	4	16	1	16		0.4096	6.5536	
								1	**9.9264**	
5 items	5	0	80	-60	1		60	0.00032	0.0192	
	4	1	65	-45	5		45	0.0064	0.288	
	3	2	50	-30	10		30	0.0512	1.536	
	2	3	35	-15	10		15	0.2048	3.072	
	1	4	20	0	5	0		0.4096	0	
	0	5	5	15	1	15		0.32768	4.9152	
								1	**9.8304**	
6 items	6	0	96	-76	1		76	0.000064	0.004864	
	5	1	81	-61	6		61	0.001536	0.093696	
	4	2	66	-46	15		46	0.01536	0.70656	
	3	3	51	-31	20		31	0.08192	2.53952	
	2	4	36	-16	15		16	0.24576	3.93216	
	1	5	21	-1	6		1	0.393216	0.393216	
	0	6	6	14	1	14		0.262144	3.670016	
					64			1	**11.340032**	

Table 6: 2-job assignment results for selection of 7, 8, 9 and 10 items

Assign	Item 1	Item 2	Total	Space	Distributn	C-Cmax	Cmax-C	Probability	Summation
7 items	7	0	112	-92	1		**92**	0.0000128	0.0011776
	6	1	97	-77	7		**77**	0.0003584	0.0275968
	5	2	82	-62	21		**62**	0.0043008	0.2666496
	4	3	67	-47	35		**47**	0.028672	1.347584
	3	4	52	-32	35		**32**	0.114688	3.670016
	2	5	37	-17	21		**17**	0.2752512	4.6792704
	1	6	22	-2	7		**2**	0.3670016	0.7340032
	0	7	7	13	1	**13**		0.2097152	2.7262976
					128			1	**13.4525952**
8 items	8	0	128	-108	1		**108**	0.00000256	0.00027648
	7	1	113	-93	8		**93**	8.192E-05	0.00761856
	6	2	98	-78	28		**78**	0.00114688	0.08945664
	5	3	83	-63	56		**63**	0.00917504	0.57802752
	4	4	68	-48	70		**48**	0.0458752	2.2020096
	3	5	53	-33	56		**33**	0.14680064	4.84442112
	2	6	38	-18	28		**18**	0.29360128	5.28482304
	1	7	23	-3	8		**3**	0.33554432	1.00663296
	0	8	8	12	1	**12**		0.16777216	2.01326592
					256			1	**16.02653184**
9 items	9	0	144	-124	1		**124**	0.000000512	6.3488E-05
	8	1	129	-109	9		**109**	0.000018432	0.002009088
	7	2	114	-94	36		**94**	0.000294912	0.027721728
	6	3	99	-79	84		**79**	0.002752512	0.217448448
	5	4	84	-64	126		**64**	0.016515072	1.056964608
	4	5	69	-49	126		**49**	0.066060288	3.236954112
	3	6	54	-34	84		**34**	0.176160768	5.989466112
	2	7	39	-19	36		**19**	0.301989888	5.737807872
	1	8	24	-4	9		**4**	0.301989888	1.207959552
	0	9	9	11	1	**11**		0.134217728	1.476395008
					512			1	**18.95279002**
10 items	10	0	160	-140	1		**140**	1.024E-07	0.000014336
	9	1	145	-125	10		**125**	0.000004096	0.000512
	8	2	130	-110	45		**110**	7.3728E-05	0.00811008
	7	3	115	-95	120		**95**	0.000786432	0.07471104
	6	4	100	-80	210		**80**	0.005505024	0.44040192
	5	5	85	-65	252		**65**	0.026424115	1.717567488
	4	6	70	-50	210		**50**	0.088080384	4.4040192
	3	7	55	-35	120		**35**	0.201326592	7.04643072
	2	8	40	-20	45		**20**	0.301989888	6.03979776
	1	9	25	-5	10		**5**	0.268435456	1.34217728
	0	10	10	10	1	**10**		0.107374182	1.073741824
					1024			1	**22.14748365**

5.5 Job Assignment Scenarios

A job assignment may consist of several scenarios. The scenarios would be under-utilized most of the time at the start of job assignments. After the initial steps, the assignment of an additional job to any of the scenarios may result either in that scenario to be still under-utilized or over-utilized.

An example of a typical job assignment with 5 scenarios, S_1, S_2, S_3, S_4, S_5, that are under-utilized is shown as follows:

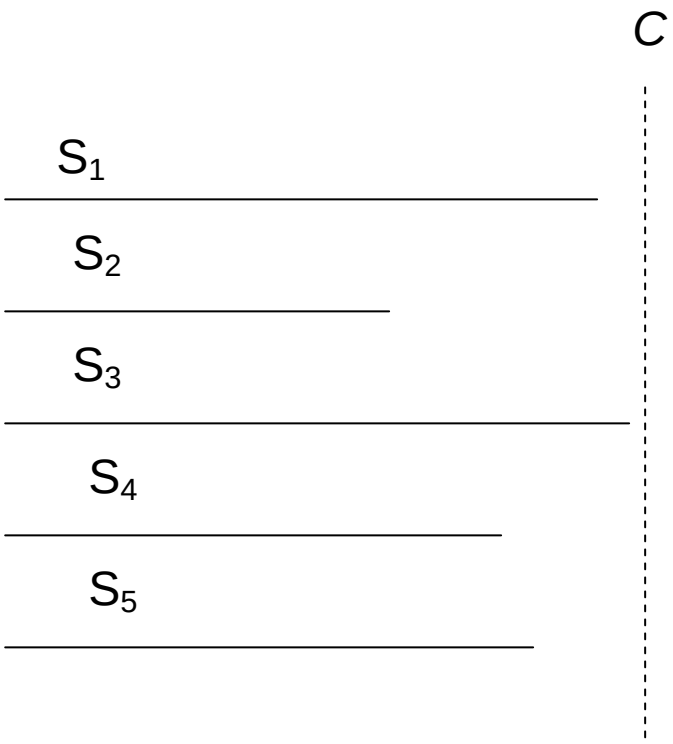


Figure 2: Typical Job Assignment Scenarios

The current objective function for this setup is

$$S_1\lambda_1 d_1 + S_2\lambda_1 d_2 + S_3\lambda_1 d_3 + S_4\lambda_1 d_4 + S_5\lambda_1 d_5 \qquad (5.6)$$

$d_i = C - c^i_{\max}$ for i = 1 to 5

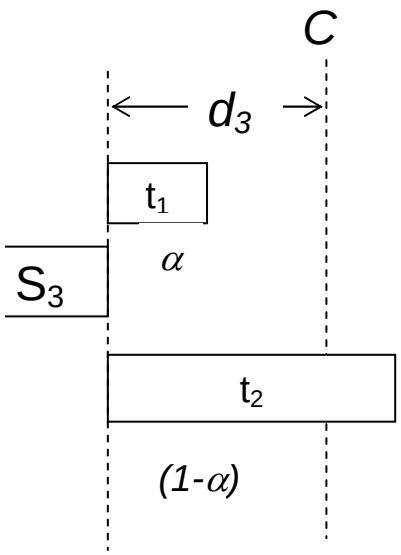


Figure 3: Addition of a job to the current assignment of S_3 with n jobs

To prove the convexity of the value function, we first analyze an example corresponding to scenario 3. Assume we have assigned n jobs and we plan to assign the $n+1^{\text{th}}$ job. For instance, supposing job n+1 is added to S_3, this makes the objective value of this scenario to be

$$\begin{aligned}&\max\{\lambda_1 S_3\alpha(d_3 - t^1_{n+1}), \lambda_2 S_3\alpha(t^1_{n+1} - d_3)\}\\&+\max\{\lambda_1 S_3(1-\alpha)(d_3 - t^2_{n+1}), \lambda_2 S_3(1-\alpha)(t^2_{n+1} - d_3)\}\end{aligned} \tag{5.7}$$

where,

t^1_{n+1} = processing time of job n+1 with probability α

t^2_{n+1} = processing time of job n+1 with probability 1-α

Assuming all assignments on S_3 exceed available time of resources, this leads to

$$\lambda_2 S_3\alpha(t^1_{n+1} - d_3) + \lambda_2 S_3(1-\alpha)(t^2_{n+1} - d_3) \tag{5.8}$$

In general, if for both scenarios after the assignment of the n+1th job, the resource limit (time limit C) is not reached, we have $f^{s_3}(n+1) - f^{s_3}(n) = -\lambda_1 S_3 E[T]$. (5.9)

If $f^{s_3}(n) \geq C$, then we have $f^{s_3}(n+1) - f^{s_3}(n) = \lambda_2 S_3 E[T]$. (5.10)

Otherwise, if $f^{s_3}(n) < C$, we analyze the following cases:

(1) if for both scenarios after assignment of the n+1th job, the resource limit (time limit C) is not reached, we have $f^{s_3}(n+1) - f^{s_3}(n) = -\lambda_1 S_3 E[T]$. (5.11)

(2) if only one assignment exceeded the capacity and the other did not, without loss of generality, we assume $t_{n+1}^1 < t_{n+1}^2$. Then,

$$f^{s_3}(n+1) - f^{s_3}(n) = \lambda_1 S_3 \alpha (d_3 - t_{n+1}^1) + \lambda_2 S_3 (1-\alpha)(t_{n+1}^2 - d_3) - \lambda_1 S_3 d_3. \quad (5.12)$$

(3) if both assignments exceed the capacity, then we have

$$f^{s_3}(n+1) - f^{s_3}(n) = \lambda_2 S_3 \alpha (t_{n+1}^1 - d_3) + \lambda_2 S_3 (1-\alpha)(t_{n+1}^2 - d_3) - \lambda_1 S_3 d_3. \quad (5.13)$$

5.6 Convexity of Cost Function

It was noticeable during experiment and testing that the cost function has convexity properties, hence in this section, we prove that the cost function, *f(n)*, is convex.

To prove convexity, we need to show that for each scenario, we have

$$\frac{\partial f^s(n)}{\partial n} - \frac{\partial f^s(n-1)}{\partial n} \geq 0, \quad (5.14)$$

this is equivalent to proving that

$$f^s(n+2) - f^s(n+1) \geq f^s(n+1) - f^s(n) \geq f^s(n) - f^s(n-1). \quad (5.15)$$

Note here, if $f^s(n) \geq C,$ then we have increment of inserting each additional job after job *n* to be $\lambda_2 SE[T]$. If $f^s(n) < C,$ then we have increment of inserting each additional job before job *n* to be $-\lambda_1 SE[T],$ which is less than $\lambda_2 SE[T]$. The conclusion holds for both ends.

In the following, we only need to prove that the conclusion holds for steps in-between. Without loss of generality, we can assume $f^s(n) < C.$ After adding one additional job, we only need to consider two cases.

Case 1: We have the total finish time of one scenario larger than *C* and the total finishing time of the other scenario smaller than *C*, as shown in Figure 3. Under this case, we have

$$f^s(n+1) - f^s(n) = -\lambda_1 S\ \alpha t_1 + \lambda_2 S\ (1-\alpha)(t_2 - d_3) - \lambda_1 S\ (1-\alpha) d_3. \tag{5.16}$$

Note here, for notation brevity, we use t_1 instead of t_{n+1}^1 and t_2 instead of t_{n+1}^2.

Then, we can observe that,

$$f^s(n+1) - f^s(n) = -\lambda_1 S\alpha t_1 + \lambda_2 S(1-\alpha)(t_2 - d_3) - \lambda_1 S(1-\alpha) d_3 \geq$$

$$-\lambda_1 S\alpha t_1 + \lambda_1 S(1-\alpha)(d_3 - t_2) - \lambda_1 S(1-\alpha) d_3 = -\lambda_1 SE[T]. \tag{5.17}$$

Similarly, we can observe that

$$f^s(n+1) - f^s(n) = -\lambda_1 S\ \alpha t_1 + \lambda_2 S\ (1-\alpha)(t_2 - d_3) - \lambda_1 S\ (1-\alpha) d_3 \leq$$

$$\lambda_2 S(1-\alpha) t_2 \leq \lambda_2 S(1-\alpha) t_2 + \lambda_2 S\alpha t_2 = \lambda_1 SE[T]. \tag{5.18}$$

Case 2: We have the total completion times of both realizations larger than *C*. Under this case, we have

$$f^s(n+1) - f^s(n) = \lambda_2 S\alpha(t_1 - d_3) + \lambda_2 S(1-\alpha)(t_2 - d_3). \tag{5.19}$$

It is easy to see that

$$f^s(n+1) - f^s(n) = \lambda_2 S\alpha(t_1 - d_3) + \lambda_2 S(1-\alpha)(t_2 - d_3) \geq 0 \geq -\lambda_1 SE[T] \tag{5.20}$$

and

$$\begin{aligned} &f^s(n+1) - f^s(n) = \lambda_2 S\alpha(t_1 - d_3) + \lambda_2 S(1-\alpha)(t_2 - d_3) \\ &\leq \lambda_2 S\alpha t_1 + \lambda_2 S(1-\alpha)t_2 = \lambda_2 SE[T]. \end{aligned} \tag{5.21}$$

This conclusion also holds.

In the following, we only need to show that

$$f^s(n+2) - f^s(n+1) \geq f^s(n+1) - f^s(n) \tag{5.22}$$

For the above case 1, since $f^s(n+2) - f^s(n+1) \geq f^s(n+1) - f^s(n)$ is obvious for case 2.

In order to show $f^s(n+2) - f^s(n+1) \geq f^s(n+1) - f^s(n)$ holds for case 1, if $2t_1 - d_3 > 0,$ then we have

$$\begin{aligned} &f^s(n+2) - f^s(n+1) = \lambda_2 S\alpha(1-\alpha)t_1 + \lambda_2 S(1-\alpha)(1-\alpha)t_2 + \lambda_2 S\alpha\alpha(2t_1 - d_3) \\ &+ \lambda_2 S\alpha(1-\alpha)(t_1 + t_2 - d_3) - \lambda_1 S\alpha(d_3 - t_1) \\ &= \lambda_2 S\alpha(2t_1 - d_3) + \lambda_2 S(1-\alpha)t_2 - \lambda_1 S\alpha(d_3 - t_1) \end{aligned} \tag{5.23}$$

Under this case, in order to show $f^s(n+2) - f^s(n+1) \geq f^s(n+1) - f^s(n),$ we only need to prove that

$$\begin{aligned} &\lambda_2 S\alpha(2t_1 - d_3) + \lambda_2 S(1-\alpha)t_2 - \lambda_1 S\alpha(d_3 - t_1) \\ &\geq -\lambda_1 S\alpha t_1 + \lambda_2 S(1-\alpha)(t_2 - d_3) - \lambda_1 S(1-\alpha)d_3. \end{aligned} \tag{5.24}$$

Since $2t_1 - d_3 \geq 0$ according to our assumption and $\lambda_2 S(1-\alpha)d_3 \geq 0,$ we only need to prove that $\lambda_1 S\alpha t_1 - \lambda_1 S\alpha(d_3 - t_1) \geq -\lambda_1 S(1-\alpha)d_3.$ (5.25)

It is equivalent to prove $\lambda_1 S\alpha(2t_1 - d_3) \geq 0 \geq -\lambda_1 S(1-\alpha)d_3.$ (5.26)

It is easy to see that the above inequality holds since $2t_1 \geq d_3$ and $\lambda_1 S(1-\alpha)d_3 \geq 0.$

Thus, the conclusion holds.

If $2t_1 < d_3$, then we have

$$f^s(n+2) - f^s(n+1) = \lambda_2 S\alpha(1-\alpha)t_1 + \lambda_2 S(1-\alpha)(1-\alpha)t_2 - \lambda_1 S\alpha\alpha t_1 + \lambda_2 S\alpha(1-\alpha)(t_1 + t_2 - d_3) - \lambda_1 S\alpha(1-\alpha)(d_3 - t_1). \quad (5.27)$$

We need to prove that

$$\lambda_2 S\alpha(1-\alpha)t_1 + \lambda_2 S(1-\alpha)(1-\alpha)t_2 - \lambda_1 S\alpha\alpha t_1 + \lambda_2 S\alpha(1-\alpha)(t_1 + t_2 - d_3) - \lambda_1 S\alpha(1-\alpha)(d_3 - t_1) \geq -\lambda_1 S\alpha t_1 + \lambda_2 S(1-\alpha)(t_2 - d_3) - \lambda_1 S\alpha(1-\alpha)d_3 \quad (5.28)$$

That is, we need to prove

$$\lambda_2 S\alpha(1-\alpha)t_1 - \lambda_1 S\alpha\alpha t_1 + \lambda_2 S\alpha(1-\alpha)(t_1 - d_3) - \lambda_1 S\alpha(1-\alpha)(d_3 - t_1) \geq -\lambda_1 S\alpha t_1 - \lambda_2 S(1-\alpha)d_3 - \lambda_1 S(1-\alpha)d_3. \quad (5.29)$$

Thus, we want to show

(i) $\lambda_2 S\alpha(1-\alpha)t_1 + \lambda_2 S\alpha(1-\alpha)(t_1 - d_3) + \lambda_2 S(1-\alpha)d_3 \geq 0$ (5.30)

and

(ii) $\lambda_1 S\alpha t_1 - \lambda_1 S\alpha\alpha t_1 + \lambda_1 S(1-\alpha)d_3 - \lambda_1 S\alpha(1-\alpha)(d_3 - t_1) \geq 0$ valid. (5.31)

Since $d_3 \geq \alpha(d_3 - t_1)$ and $\alpha \geq \alpha^2$, (ii) is valid.

We also have

$$\alpha t_1 + \alpha(t_1 - d_3) + d_3 = 2\alpha t_1 + (1-\alpha)d_3 \geq 2\alpha t_1 + 2(1-\alpha)t_1 = 2t_1 \geq 0. \quad (5.32)$$

Therefore, (i) holds.

We also can conclude the function is convex.

5.7 The *n*-job *m*-processor Problem

This is the extension of the *n*-job-1-processor problem discussed earlier. This problem involves *m* number of inspectors, and *n* number of packages with *l* number of possible outcomes. The objective of the problem is to minimize the expected penalty cost for all inspectors. Each job will go through pre-scan to visualize the processing time of the job. Therefore, the processing time of each type of package is known before assignment, and dependent on the probability of the selection of that package. Two kinds of problems were formulated in this category. One is to find the maximum number of packages that could be assigned within a given time limit. The other is to minimize the total penalty cost for all inspectors for a given number of packages.

5.7.1 Maximize the Number of Packages and Minimize Penalty Cost

Packages are assigned to the inspectors based on the amount of resources available. The next package is assigned to the inspector with the largest available resources. This is done to achieve a sequential reduction of available resources to each inspector. This will in turn lead to the minimization of penalty incurred. There is a penalty for both under-utilization and over-utilization of resources. In this problem, only the under-utilization penalty was considered.

The following notations were used throughout the definition of this problem:

n = total number of packages

m = total number of inspectors

l = possible outcomes for all packages

p_{jl} = inspection time of package j with outcome l

α_l = probability associated with p_{jl}

C = total available resources for each inspector

λ_1 = unit penalty cost for under-utilization

λ_2 = unit penalty cost for over-utilization

c^i = competition time of the last job in inspection i

5.8 The *n*-job *m*-processor Problem Formulation

Objective function of this problem is to minimize the expected penalty cost for all inspectors. This can be formulated as below:

$$\text{min. } \sum_{i=1}^{m}\left(\sum_{S_1=1}^{l}\sum_{S_2=1}^{l}...\sum_{S_n=1}^{l}\alpha_{s_1}\alpha_{s_2}\alpha_{s_n}\{\lambda_1\max(C-c^i,0)+\lambda_2\max(c^i-C,0)\}\right) \qquad (5.33)$$

$$\text{st.} \quad \sum_{i=1}^{m} x_{ij}=1, \qquad j=1,...,n.$$

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

$$c^i=\sum_{j=1}^{n}x_{ij}p_{jS_j} \qquad i=1,...,m.$$

x_{ij} = package j is assigned to inspector i

p_{jl} = inspection time of package j with outcome l

$c^i = x_{i1}p_{1S_1}+x_{i2}p_{2S_2}+....+x_{in}p_{nS_n}$

Since only the under-utilization of resources was considered for this problem, the formulation can be simplified as:

$$\text{min. } \sum_{i=1}^{m}\left(\sum_{S_1=1}^{l}\sum_{S_2=1}^{l}\ldots\sum_{S_n=1}^{l}\alpha_{s_1}\alpha_{s_2}\alpha_{s_n}\lambda\ \max(C-c^i,0)\right) \tag{5.34}$$

This formulation is subject to the same set of constraints as the one above, but with only the under-utilization penalty cost λ.

5.9 The Solution Method for the *n*-job *m*-processor Problem

We use the sampling approximation average method to solve the problem. To obtain the average value, we take samples and obtain the average value of these samples as the approximation of the objective function value. The solution of each sample will involve the assignment of packages to inspectors until all inspectors have been fully utilized. The Largest Unutilized Capacity First (*LUCF)* rule was again utilized in the solution to solve each sample.

Each sample can be solved using the following algorithmic steps:

Initialize $P = N$, $R = M$, $c^i = 0$ (utilized space), C (available resources). For notation brevity, we let p_j represent the realized processing time p_{jsj} for some s_j based on the sampling result.

(1) Pick a package j from P

(2) Select an inspector i with min (c^i)

(3) If $c^i + p_j > c$

(a) Remove inspector i from R

(b) Go to Step (2)

(4) Otherwise assign package j to inspector i, remove package j from P and update $c^i = c^i + p_j$

(5) If either R or P is nonempty, go to Step (1).

(6) Otherwise, terminate the algorithm.

These steps are represented by the flowchart below:

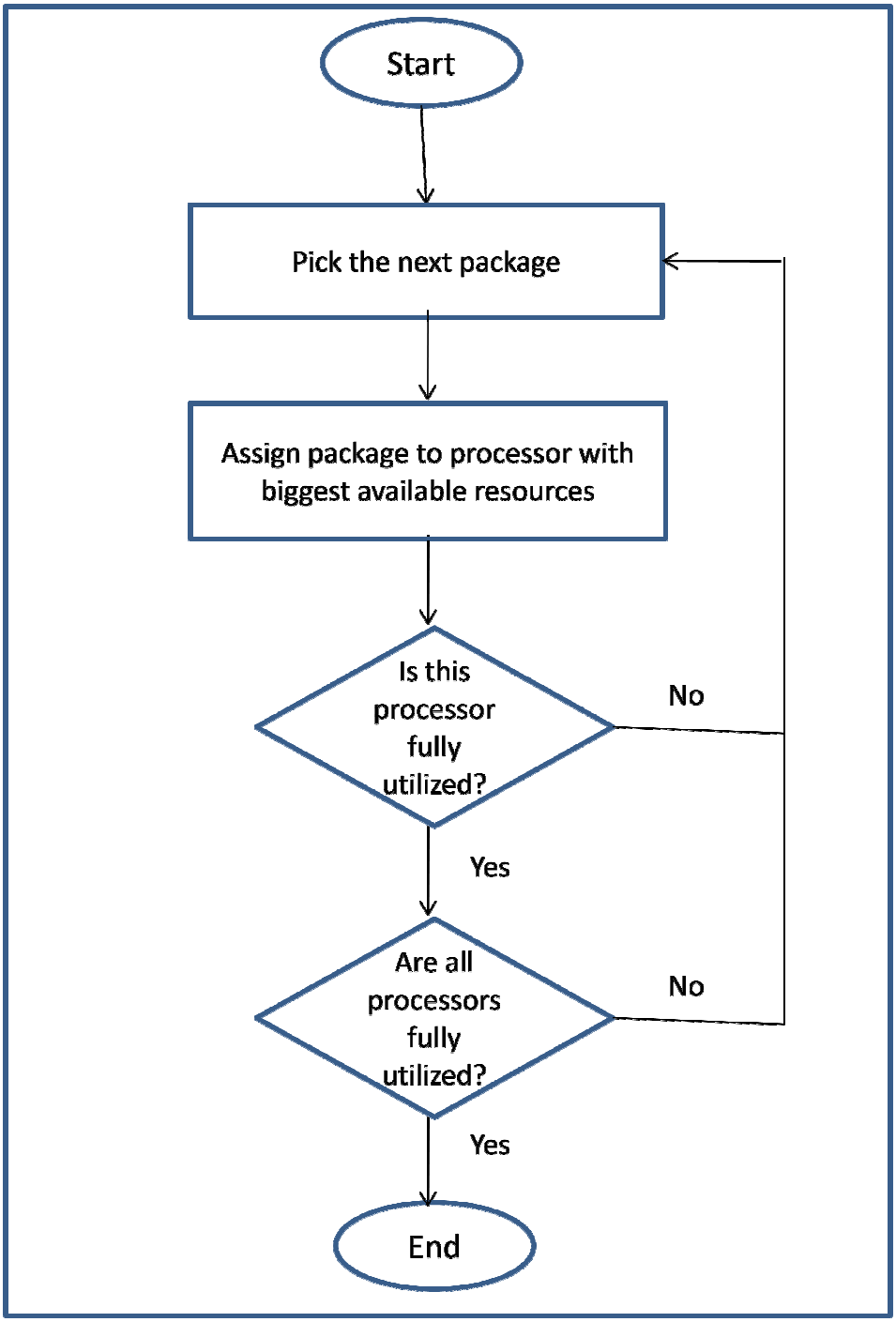


Figure 4: Algorithmic steps of the 1st SKPPC Problem

5.10 Numerical Example Data Generation

We test the case that each job has two possible realizations. For instance, l=2. Uncorrelated package were generated with processing times p_j uniformly distributed in [10, 100], and capacities c_i uniformly distributed in $\left[0.4\sum_{j=1}^{n} p_j / m, 0.6\sum_{j=1}^{n} p_j / m\right]$ for i = 1, m-1. The capacity of the m^{th} knapsack was set to $c_m = \left[0.5\sum_{j=1}^{n} p_j - \sum_{i=1}^{m-1} c_j\right]$. Note here, p_{j} represents the expected processing time. Based on this, we also assume p_{jsj} uniformly distributed in [10, 100]. The average of all capacities was then calculated for equal resources amongst all inspectors. A random number is generated to determine p_{j} to be p_{jsj} for some s_{j}=1 or 2. If the random number is less than α_{s_1}, then $p_{\text{j}} = p_{\text{j1.}}$ Otherwise, $p_{\text{j=}}\ p_{\text{j2.}}$ All data generated were made integers except for the probabilistic values. Finally, we set λ equal to 1.

The following conditions must be satisfied for all formulations:

(1) $p_{jsj} > 0$ and integers for all j

(2) $c_i > 0$ and integer for all i

(3) $\min_{j,s_j}\{p_j s_j\} \le \min_i\{c_i\}$

(4) $\max_{j,s_j}\{p_j s_j\} \le \max_i\{c_i\}$

(5) $\sum_{i=1}^{n} p_j > \max_i \{ c_i \}$

5.11 Numerical Example Results

As described in the previous section, each package has two possible processing times with each having its own probability of being assigned. A number was generated between 0 and 1 to determine the processing time of each job.

The proposed *LUCF* was used to assign job after pre-scan. Assignment was stopped once none of the packages left can be accepted by any inspector without being over-utilized. The penalty was calculated for each package/inspector combinations. One hundred samples were computed for each combination and the average penalty and capacity utilization were calculated. We tested cases with package sizes to be 25, 50, 100, and 200 and they are assigned to 2, 3, 5, and 10 inspectors respectively.

Table 7 shows the results obtained for the various package/inspector combinations. The actual expected penalty costs are recorded in the table. It was observed that the larger the ratio *m/n*, the better the results produced.

Table 7: Results of 1st SKPPC Problem.

# of Inspectors m	# of Packages n	Solution	
		Penalty Cost	% Utilization
2	25	1.66	97.49
	50	0.11	99.17
	100	0.00	99.63
	200	0.00	99.85
3	25	6.02	95.11
	50	0.79	98.52
	100	0.02	99.41
	200	0.00	99.77
5	25	23.50	90.69
	50	6.04	96.78
	100	0.58	98.95
	200	0.01	99.58
10	25	107.17	74.48
	50	45.42	90.77
	100	10.81	97.06
	200	1.04	99.05

Chapter 6

Inspection Problem – an SKPPC Problem

6.1 Introduction

Inspection of packages or containers at an inbound point, security check of passengers at the airports, inspection of goods and services, in general, require allocation of resources. In this section, we focused our attention on inspection of packages where each package can be classified as a high-risk and low-risk package depending on a set of factors determining the level of risk for the package. Let α_l denote the probability that a package is of type l, where l=1, 2 represent high-risk and low-risk packages, respectively. Let p_{jl} denote the processing time for package j of type l. Assuming m inspectors and C time units of maximum inspection time per inspector, the problem is to determine the number of inspectors and number of packages to be assigned to each inspector in a way that the expected value of the total inspection cost is minimized. The total inspection cost is defined as the weighted cost of the under-utilization and over-utilization of the inspectors during the inspection period, C. We next formulate the problem as a stochastic knapsack problem and discuss solution methodology.

6.2 Problem Formulation

As discussed in the previous chapter, the inspection problem can be formulated as follows:

$$\min \sum_{i=1}^{m}\left(\sum_{S_1=1}^{l}\sum_{S_2=1}^{l}\ldots\sum_{S_n=1}^{l}\alpha_{s_1}\alpha_{s_2}\alpha_{s_n}\{\lambda_1\max(C-c^i,0)+\lambda_2\max(c^i-C,0)\}\right) \tag{6.1}$$

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

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

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

where,

x_{ij} = package j is assigned to inspector i

c^i = total inspection time for inspector i under scenario S_i.

For an n-package and m-inspector inspection problem, there will be 2^n package arrival combinations with each package being either high or low-risk type. Each inspector may inspect none or all packages provided that the sum of packages inspected by all of the inspectors equal n. Let S_i denote the number of packages inspected by inspector i with probability α_i of occurrence. Then, $\alpha_{S_1}\alpha_{S_2}\ldots\alpha_{S_m}$ is the probability of a scenario. As an example, consider a 4-package, 2-inspector problem. There are 8 possible inspection sequences with 1 and 2 indicating high-risk and low-risk packages, respectively. These combinations are:

1 1 1 1
1 1 1 2
1 1 2 1
1 1 2 2
1 2 1 1
1 2 1 2
1 2 2 1
1 2 2 2

There are 5 scenarios for each inspector as shown below:

Table 8: Scenarios for 2-inspector, 4-package assignments

Scenarios	Inspector 1	Inspector 2
1	0 packages	4 packages
2	1 package	3 package
3	2 packages	2 packages
4	3 packages	1 package
5	4 packages	0 packages

For the sake of simplicity, let's refer to Scenario 1 as the scenario where inspector 2 inspects all the packages. Similarly, let's redefine the rest of the scenarios. Scenarios 4 and 5 will have same probability of occurrence and same objective function value as Scenarios 1 and 2, hence will not be separately calculated. Let the objective function value contributed by a scenario be indicated by $Z_n(j, n-j)$. This means that out of n inspected packages; j packages are inspected by inspector 1, and the rest by inspector 2.

6.3 Algorithm

An algorithm was developed to solve any instance of this problem involving any number of inspectors, any number of packages with 2-package types.

6.3.1 Notations

n = numbers of packages inspected

m = numbers of inspectors

α_1 = probability associated with assignment of package type *1*

α_2 = probability associated with assignment of package type *2*

p_1 = inspection time for package type *1*

p_2 = inspection time for package type *2*

C = time available for inspection

Z^* = optimal scenario, scenario with the minimum total inspection cost

S_i = scenario *i*

c_i = inspection cost of scenario *i*

6.3.2 Steps of the Algorithm

Step 1: Initialize the variables n, m, α_1, α_2, p_1, p_2, C, Z^*, S_i, c_i.

Step 2: Compute all possible package arrival combinations for n and all possible feasible scenarios for m.

Step 3: Calculate the penalty cost c_i for S_i.

Step 4: Record $Z_i(S_i)$ for the minimum c_i.

If $Z_i(S_i) < Z^*$, replace Z^* with $Z_i(S_i)$.

If S_i is the last scenario, go to Step 5, else go to Step 3.

Step 5: Record $Z_i(S_i)$.

All computations of the algorithm were performed in MATLAB. The number of packages to be inspected, number of inspectors, probabilities of selection, processing times of packages, time available to inspectors, and the utilization penalties were all initially specified at the beginning of the MATLAB code.

The unique combinations of package assignment was then computed, and stored in an array, with the probability for each assignment.

The total penalty cost for all scenarios was then calculated, and the table of results generated. The data was then exported into EXCEL for the graphical representation.

The problems of assignment of 15 packages were solved within minutes. This was repeated for different values of the variables. Most graphs for data representation were produced in EXCEL, with just a handful in MATLAB. This was because of the easier data manipulation in EXCEL to quickly adjust a graphical output to showcase a different data set.

6.4 Numerical Example

The under-utilization penalty, λ_1, and the over-utilization penalty, λ_2 are constants such that $\lambda_1 + \lambda_2 = 1$.

An inspection problem considering two-package types with inspection time, p_1 and p_2 of 16 and 1 and probabilities α_1 and α_2 of selection 0.2 and 0.8, respectively was formulated, solved using MATLAB and EXCEL and results tabulated as shown in Table 9. The number of packages assigned ranged from 3 to 10, although the program is able to give results for any number of packages. The table below shows the results obtained for five values of the utilization penalties with capacity (inspection time) of 20 time units. The results in the table were the best objective function value for each scenario. Note that, the best solution for each parameter value combinations occur, when the packages are distributed equally among the inspectors. This is logical since

one does not know the type of the package to be inspected until the inspection is done on the package and that the inspectors have same amount of inspection time available to them. Hence both the packages and inspectors are considered indistinguishable by this problem.

Table 9: Penalty Cost Results for Some Scenarios with different utilization penalty

		p1=16,p2=1,prob1=0.2,prob2=0.8 capacity =20				
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z_3(1,2)	**3**	3.2800	6.0800	8.8800	11.6800	14.4800
Z_4(2,2)	**4**	3.3600	5.7600	8.1600	10.5600	12.9600
Z_5(2,3)	**5**	3.9520	5.9520	7.9520	9.9520	11.9520
Z_6(3,3)	**6**	4.5440	6.1440	7.7440	9.3440	10.9440
Z_7(3,4)	**7**	5.6352	6.8352	8.0352	9.2352	10.4352
Z_8(4,4)	**8**	6.7264	7.5264	8.3264	9.1264	9.9264
Z_9(4,5)	**9**	8.2784	8.6784	9.0784	9.4784	9.8784
Z_{10}(5,5)	**10**	9.8304	9.8304	9.8304	9.8304	9.8304

From the table above, the minimum penalty cost for each problem as a function of the penalty cost is highlighted. The scenario with minimum penalty cost suggested that 10 packages would yield minimum penalty cost for a problem with equal utilization value. Assignment of five packages to each inspector will be the best assignment for this case.

Other problems were formulated and solved for various combinations of the input parameter values. Table 10 contains results with 2, 3 and 4 inspectors. Figure 5 illustrates graphs for the expected total inspection costs as a function of the number of packages inspected when the under-utilization and over-utilization costs are penalized equally. It can be seen from the graphs that the expected total penalty cost increases as the number of inspectors increase. For the 2-inspector problem, the minimum penalty cost of 9.8304 came from scenario $Z_{10}(5, 5)$. The minimum penalty cost of 14.7456

resulted from scenario $Z_{15}(5, 5, 5)$ for the 3-inspector problem. The question one can pose is; if 15 packages are to be inspected during a time period, is it better to have 2 inspectors or 3 inspectors? In general, what is the optimal number of inspectors needed as a function of number of packages to be inspected?

Table 10: Total cost as a function of *n* and *m*

p1 = 16, p2 = 1	**prob1 = 0.2, prob2 = 0.8**	**k = 20**	**λ1 = 0.1**
n	**m=2**	**m=3**	**m=4**
3	14.4800	24.0000	34.0000
4	12.9600	22.4800	32.0000
5	11.9520	20.9600	30.4800
6	10.9440	19.4400	28.9600
7	10.4352	18.4320	27.4400
8	9.9264	17.4240	25.9200
9	9.8784	16.4160	24.9120
10	9.8304	15.9072	23.9040
11	10.5852	15.3984	22.8960
12	11.3400	14.8896	21.8880
13	12.3963	14.8416	21.3792
14	13.4526	14.7936	20.8704
15	14.7396	14.7456	20.3616
16	16.0265	15.5004	19.8528
17	17.4897	16.2552	19.8048
18	18.9528	17.0100	19.7568
19	20.5501	18.0663	19.7088
20	22.1475	19.1226	19.6608

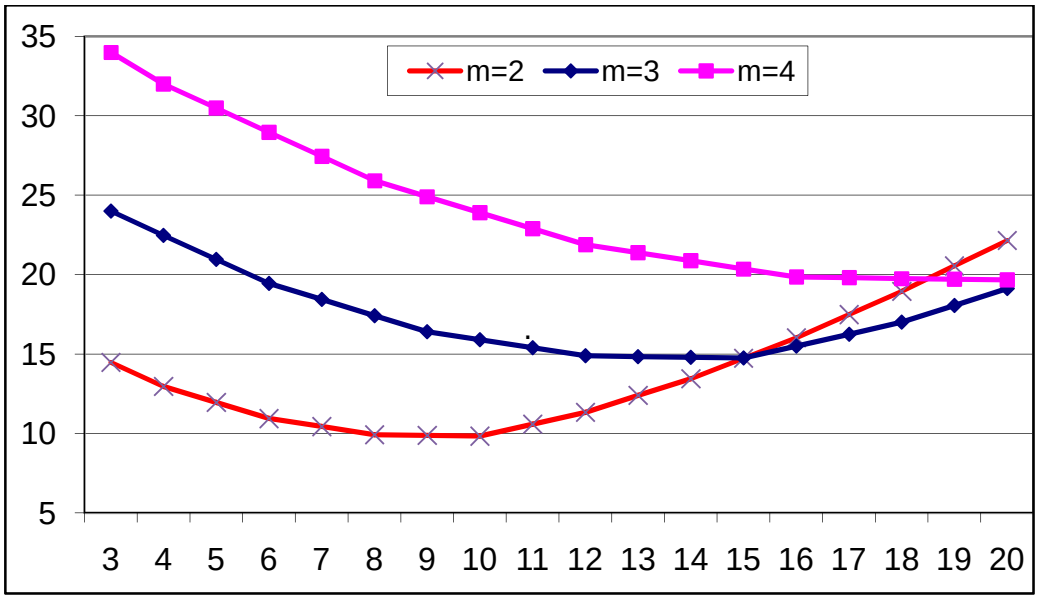


Figure 5: Expected total inspection cost versus *n* and *m*

It can be seen from Figure 5 that the expected total penalty cost is convex as a function of *n*. As *m* increases the value of *n* which gives the minimum expected total inspection cost increases. For the 2-inspector problem, the minimum penalty cost of 9.8304 came from scenario Z_{10} (5, 5). The minimum penalty cost of 14.7456 resulted from scenario Z_{15} (5, 5, 5) for the 3-inspector problem. Similarly, Z_{20} (5, 5, 5, 5) gives the minimum cost for the 4-inspector problem. The optimal number of jobs to be assigned to each inspector is 5. For the example displayed by Table 10 and Figure 5, if number of items to be inspected during an inspection period is less than 15, then using two inspectors will give the minimum expected cost. If the number of packages inspected is more than 20, then 4 inspectors will give the minimum expected cost. One needs to run the algorithm with more than 20 packages to determine when it is best to add another inspector.

6.5 Experimental Results

Experiments were run with various values of processing times, capacities (maximum inspector time), utilization penalty, and proportion of high-risk and low-risk items. The next sections discuss results for each parameter variation.

6.5.1 Changes in Capacities

The experiments were run by varying the maximum available inspector time, also referred to as capacity, *C*. Each inspector was assumed to have the same capacity, ranging from 20 to 50 time units. These capacity values were chosen arbitrarily. The following tables and graphs illustrate results for *C*= 20, 30, 40, and 50.

Table 11: Capacity of 20 results for 5 values of λ_1

p1=16,p2=1,prob1=0.2,prob2=0.8 capacity =20						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	3.2800	6.0800	8.8800	11.6800	14.4800
Z4(2,2)	**4**	3.3600	5.7600	8.1600	10.5600	12.9600
Z5(2,3)	**5**	3.9520	5.9520	7.9520	9.9520	11.9520
Z6(3,3)	**6**	4.5440	6.1440	7.7440	9.3440	10.9440
Z7(3,4)	**7**	5.6352	6.8352	8.0352	9.2352	10.4352
Z8(4,4)	**8**	6.7264	7.5264	8.3264	9.1264	9.9264
Z9(4,5)	**9**	8.2784	8.6784	9.0784	9.4784	9.8784
Z10(5,5)	**10**	9.8304	9.8304	9.8304	9.8304	9.8304
Z11(5,6)	**11**	12.1852	11.7852	11.3852	10.9852	10.5852
Z12(6,6)	**12**	14.5400	13.7400	12.9400	12.1400	11.3400
Z13(6,7)	**13**	17.1963	15.9963	14.7963	13.5963	12.3963
Z14(7,7)	**14**	19.8526	18.2526	16.6526	15.0526	13.4526
Z15(7,8)	**15**	22.7396	20.7396	18.7396	16.7396	14.7396

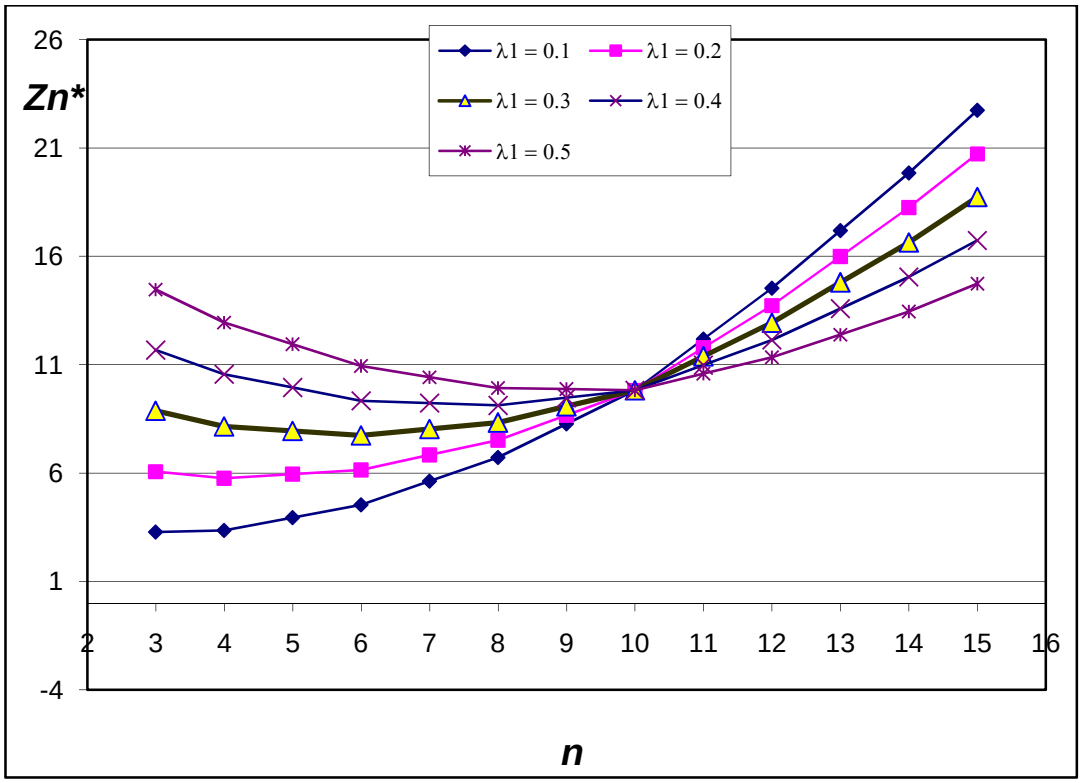


Figure 6: Graph for capacity of 20 results for 5 values of λ_1

Table 12: Capacity of 30 results for 5 values of λ_1

p1=16,p2=1,prob1=0.2,prob2=0.8 capacity =30						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	4.8800	9.6800	14.4800	19.2800	24.0800
Z4(2,2)	**4**	4.5600	8.9600	13.3600	17.7600	22.1600
Z5(2,3)	**5**	4.5120	8.5120	12.5120	16.5120	20.5120
Z6(3,3)	**6**	4.4640	8.0640	11.6640	15.2640	18.8640
Z7(3,4)	**7**	4.7872	7.9872	11.1872	14.3872	17.5872
Z8(4,4)	**8**	5.1104	7.9104	10.7104	13.5104	16.3104
Z9(4,5)	**9**	5.8432	8.2432	10.6432	13.0432	15.4432
Z10(5,5)	**10**	6.5760	8.5760	10.5760	12.5760	14.5760
Z11(5,6)	**11**	7.7184	9.3184	10.9184	12.5184	14.1184
Z12(6,6)	**12**	8.8608	10.0608	11.2608	12.4608	13.6608
Z13(6,7)	**13**	10.3899	11.1899	11.9899	12.7899	13.5899
Z14(7,7)	**14**	11.9189	12.3189	12.7189	13.1189	13.5189
Z15(7,8)	**15**	13.7993	13.7993	13.7993	13.7993	13.7993

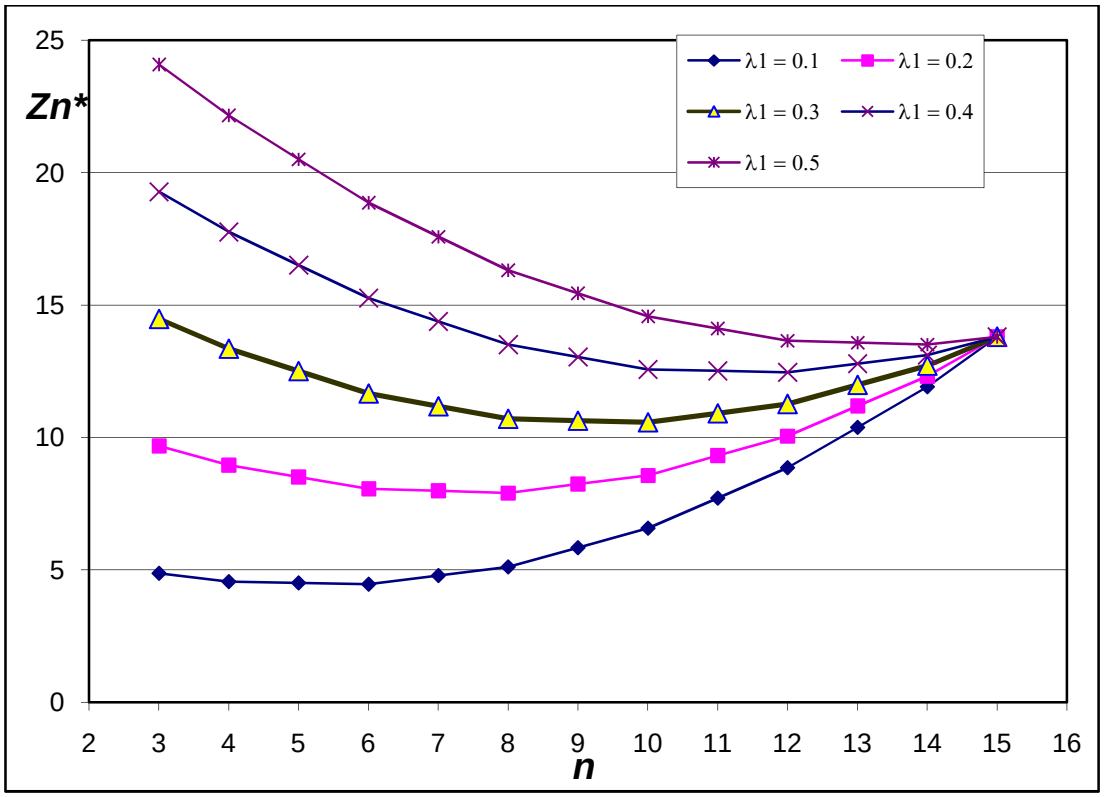


Figure 7: Graph for capacity of 30 results for 5 values of λ_1

Table 13: Capacity of 40 results for 5 values of λ_1

p1=16,p2=1,prob1=0.2,prob2=0.8 capacity =40						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	6.8000	13.6000	20.4000	27.2000	34.0000
Z4(2,2)	**4**	6.4000	12.8000	19.2000	25.6000	32.0000
Z5(2,3)	**5**	6.0640	12.0640	18.0640	24.0640	30.0640
Z6(3,3)	**6**	5.7280	11.3280	16.9280	22.5280	28.1280
Z7(3,4)	**7**	5.5328	10.7328	15.9328	21.1328	26.3328
Z8(4,4)	**8**	5.3376	10.1376	14.9376	19.7376	24.5376
Z9(4,5)	**9**	5.3536	9.7536	14.1536	18.5536	22.9536
Z10(5,5)	**10**	5.3696	9.3696	13.3696	17.3696	21.3696
Z11(5,6)	**11**	5.6518	9.2518	12.8518	16.4518	20.0518
Z12(6,6)	**12**	5.9341	9.1341	12.3341	15.5341	18.7341
Z13(6,7)	**13**	6.5194	9.3194	12.1194	14.9194	17.7194
Z14(7,7)	**14**	7.1048	9.5048	11.9048	14.3048	16.7048
Z15(7,8)	**15**	8.0125	10.0125	12.0125	14.0125	16.0125

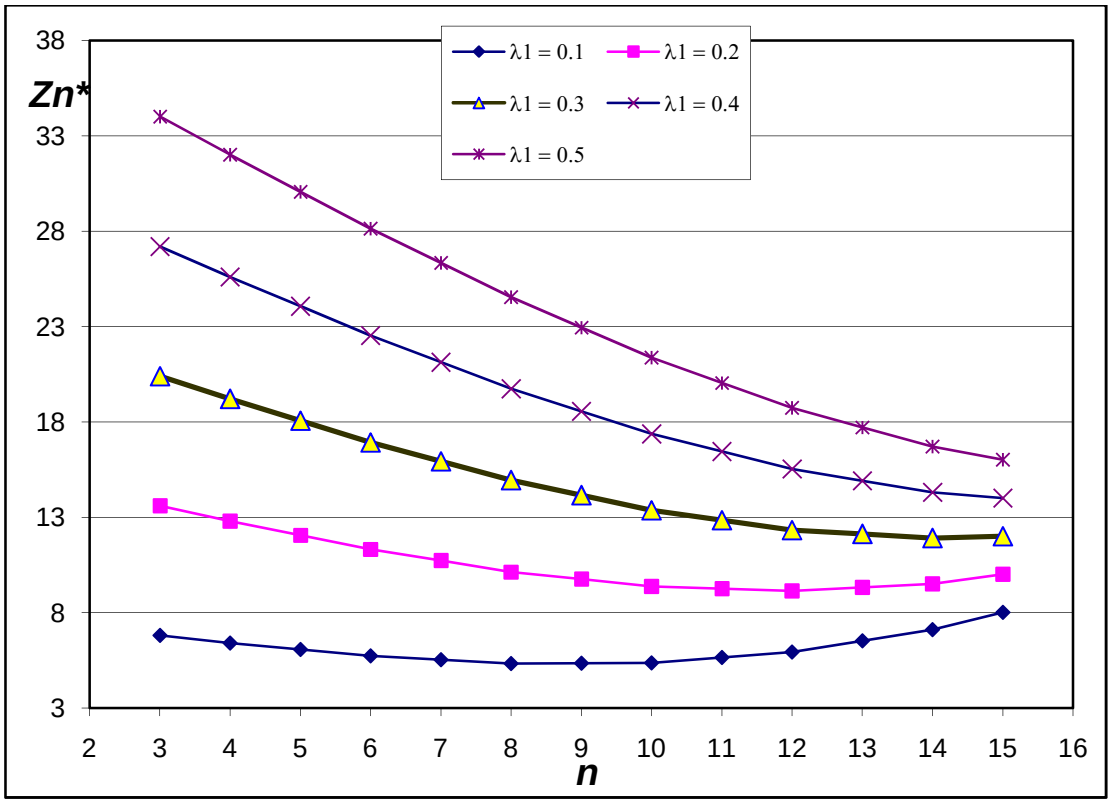


Figure 8: Graph for capacity of 40 results for 5 values of λ_1

Table 14: Capacity of 50 results for 5 values of λ_1

p1=16,p2=1,prob1=0.2,prob2=0.8 capacity =50						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	8.8000	17.6000	26.4000	35.2000	44.0000
Z4(2,2)	**4**	8.4000	16.8000	25.2000	33.6000	42.0000
Z5(2,3)	**5**	8.0000	16.0000	24.0000	32.0000	40.0000
Z6(3,3)	**6**	7.6000	15.2000	22.8000	30.4000	38.0000
Z7(3,4)	**7**	7.2224	14.4224	21.6224	28.8224	36.0224
Z8(4,4)	**8**	6.8448	13.6448	20.4448	27.2448	34.0448
Z9(4,5)	**9**	6.5280	12.9280	19.3280	25.7280	32.1280
Z10(5,5)	**10**	6.2112	12.2112	18.2112	24.2112	30.2112
Z11(5,6)	**11**	6.0838	11.6838	17.2838	22.8838	28.4838
Z12(6,6)	**12**	5.9565	11.1565	16.3565	21.5565	26.7565
Z13(6,7)	**13**	6.0503	10.8503	15.6503	20.4503	25.2503
Z14(7,7)	**14**	6.1441	10.5441	14.9441	19.3441	23.7441
Z15(7,8)	**15**	6.5014	10.5014	14.5014	18.5014	22.5014

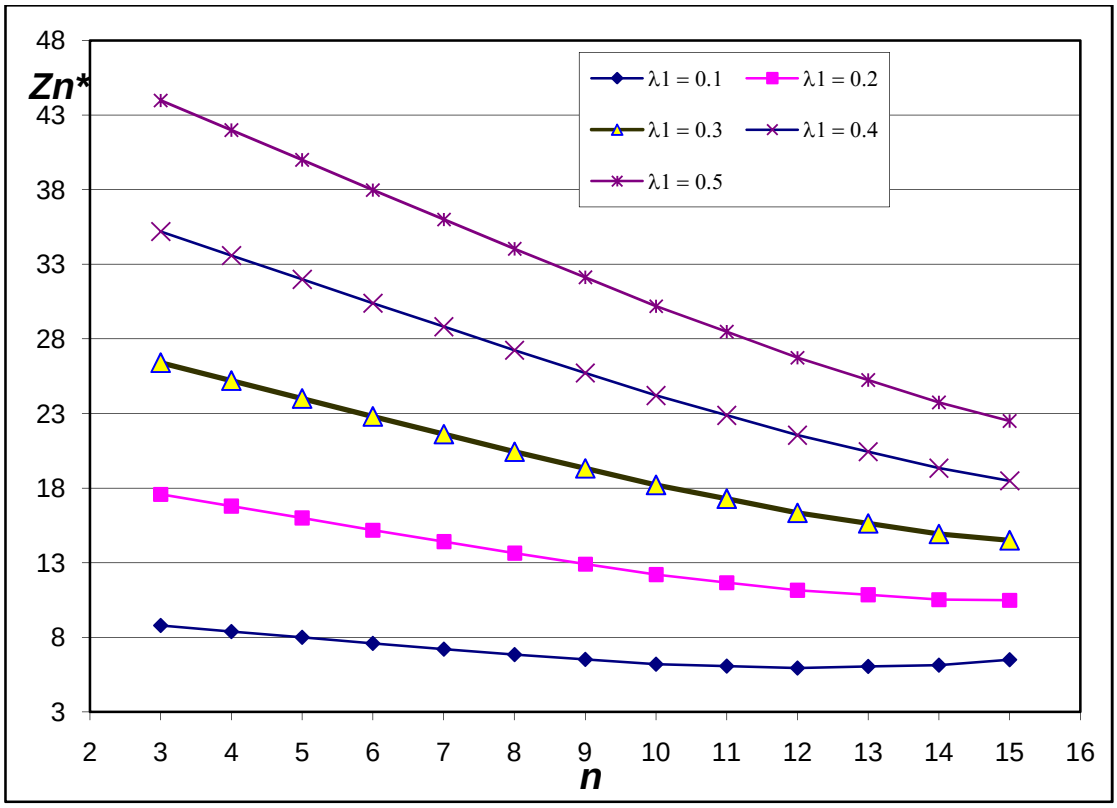


Figure 9: Graph for capacity of 50 results for 5 values of λ_1

The increase in capacity resulted in increase in the penalty cost in all cases. This was to be expected since there is more resources available to the inspectors. The resulting minimum penalty cost for each value of λ_1 has also changed. For example, the minimum penalty cost changed from 3.2800 to 4.4640 for λ_1=0.1 for capacities 20 and 30 respectively. The resulting assignment also changed from $Z_3(1,2)$ for capacity 20 to $Z_6(3,3)$ for capacity 30. Similar observations were noticed for other values.

6.5.2 Changes in Probabilities

The design of experiment was next formulated for changing values in the probabilities of selection of the 2 package types. A capacity of 20 time units was chosen, with all other variables constant. The probability of selection of package type 1

used were chosen as 0.2, 0.3, 0.4, and 0.5. The following tables and graphs resulted from that experiment.

Table 15: Package type 1 probability of 0.2 results for 5 values of λ_1

		p1=16,p2=1,capacity=20 prob1=0.2				
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	3.2800	6.0800	8.8800	11.6800	14.4800
Z4(2,2)	**4**	3.3600	5.7600	8.1600	10.5600	12.9600
Z5(2,3)	**5**	3.9520	5.9520	7.9520	9.9520	11.9520
Z6(3,3)	**6**	4.5440	6.1440	7.7440	9.3440	10.9440
Z7(3,4)	**7**	5.6352	6.8352	8.0352	9.2352	10.4352
Z8(4,4)	**8**	6.7264	7.5264	8.3264	9.1264	9.9264
Z9(4,5)	**9**	8.2784	8.6784	9.0784	9.4784	9.8784
Z10(5,5)	**10**	9.8304	9.8304	9.8304	9.8304	9.8304
Z11(5,6)	**11**	12.1852	11.7852	11.3852	10.9852	10.5852
Z12(6,6)	**12**	14.5400	13.7400	12.9400	12.1400	11.3400
Z13(6,7)	**13**	17.1963	15.9963	14.7963	13.5963	12.3963
Z14(7,7)	**14**	19.8526	18.2526	16.6526	15.0526	13.4526
Z15(7,8)	**15**	22.7396	20.7396	18.7396	16.7396	14.7396

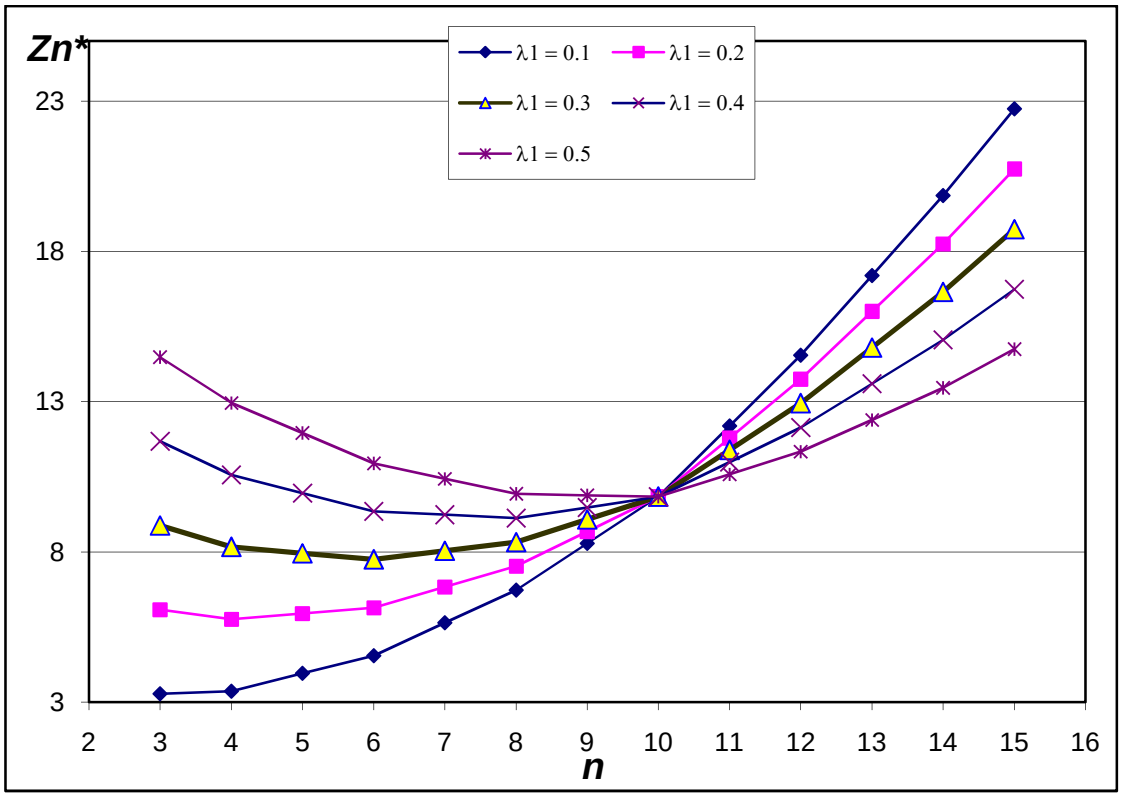


Figure 10: Graph for Package type 1 probability of 0.2 results for 5 values of λ_1

Table 16: Package type 1 probability of 0.3 results for 5 values of λ_1

		p1=16,p2=1,capacity=20 prob1=0.3				
	n	λ1 = 0.1	λ1 = 0.2	λ1 = 0.3	λ1 = 0.4	λ1 = 0.5
Z3(1,2)	3	3.4300	5.7800	8.1300	10.4800	12.8300
Z4(2,2)	4	3.9600	5.7600	7.5600	9.3600	11.1600
Z5(2,3)	5	5.5430	6.7930	8.0430	9.2930	10.5430
Z6(3,3)	6	7.1260	7.8260	8.5260	9.2260	9.9260
Z7(3,4)	7	9.6162	9.7662	9.9162	10.0662	10.2162
Z8(4,4)	8	12.1064	11.7064	11.3064	10.9064	10.5064
Z9(4,5)	9	15.3243	14.3743	13.4243	12.4743	11.5243
Z10(5,5)	10	18.5421	17.0421	15.5421	14.0421	12.5421
Z11(5,6)	11	22.6181	20.5681	18.5181	16.4681	14.4181
Z12(6,6)	12	26.6942	24.0942	21.4942	18.8942	16.2942
Z13(6,7)	13	31.0677	27.9177	24.7677	21.6177	18.4677
Z14(7,7)	14	35.4412	31.7412	28.0412	24.3412	20.6412
Z15(7,8)	15	40.0124	35.7624	31.5124	27.2624	23.0124

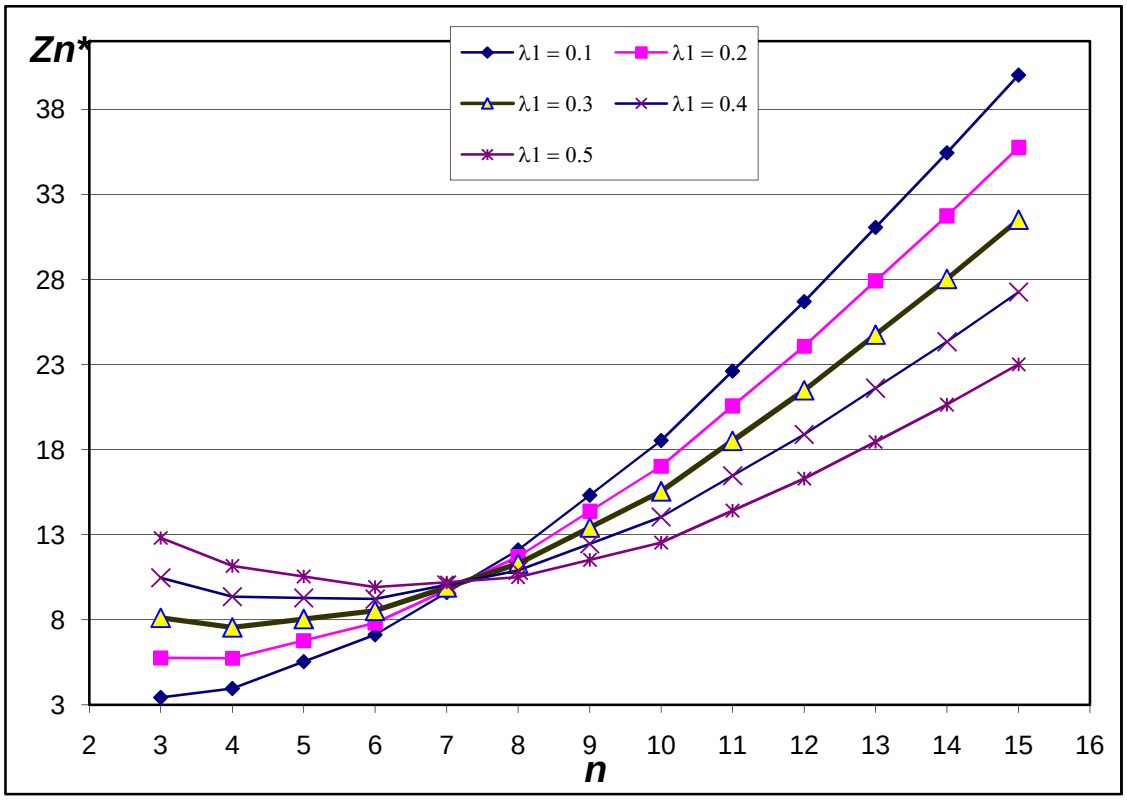


Figure 11: Graph for Package type 1 probability of 0.3 results for 5 values of λ_1

Table 17: Package type 1 probability of 0.4 results for 5 values of λ_1

	p1=16,p2=1,capacity=20 prob1=0.4					
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	3.8200	5.7200	7.6200	9.5200	11.4200
Z4(2,2)	**4**	5.0400	6.2400	7.4400	8.6400	9.8400
Z5(2,3)	**5**	7.9560	8.4560	8.9560	9.4560	9.9560
Z6(3,3)	**6**	10.8720	10.6720	10.4720	10.2720	10.0720
Z7(3,4)	**7**	15.0552	14.1552	13.2552	12.3552	11.4552
Z8(4,4)	**8**	19.2384	17.6384	16.0384	14.4384	12.8384
Z9(4,5)	**9**	24.2856	21.9856	19.6856	17.3856	15.0856
Z10(5,5)	**10**	29.3328	26.3328	23.3328	20.3328	17.3328
Z11(5,6)	**11**	35.1196	31.4196	27.7196	24.0196	20.3196
Z12(6,6)	**12**	40.9064	36.5064	32.1064	27.7064	23.3064
Z13(6,7)	**13**	46.9171	41.8171	36.7171	31.6171	26.5171
Z14(7,7)	**14**	52.9278	47.1278	41.3278	35.5278	29.7278
Z15(7,8)	**15**	59.0655	52.5655	46.0655	39.5655	33.0655

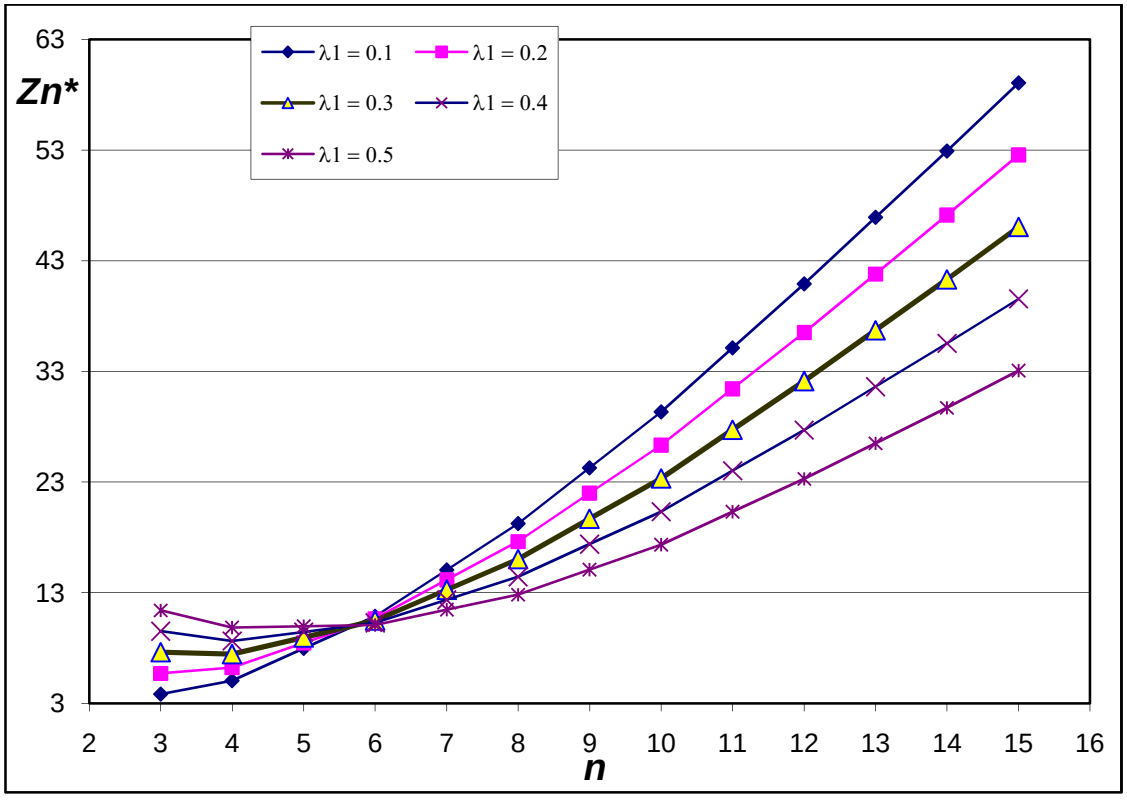


Figure 12: Graph for Package type 1 probability of 0.4 results for 5 values of λ_1

Table 18: Package type 1 probability of 0.2 results for 5 values of λ_1

	p1=16,p2=1,capacity=20 prob1=0.5					
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	4.4500	5.9000	7.3500	8.8000	10.2500
Z4(2,2)	**4**	6.6000	7.2000	7.8000	8.4000	9.0000
Z5(2,3)	**5**	11.1250	10.8750	10.6250	10.3750	10.1250
Z6(3,3)	**6**	15.6500	14.5500	13.4500	12.3500	11.2500
Z7(3,4)	**7**	21.6750	19.7250	17.7750	15.8250	13.8750
Z8(4,4)	**8**	27.7000	24.9000	22.1000	19.3000	16.5000
Z9(4,5)	**9**	34.5688	30.9188	27.2688	23.6188	19.9688
Z10(5,5)	**10**	41.4375	36.9375	32.4375	27.9375	23.4375
Z11(5,6)	**11**	48.8375	43.4875	38.1375	32.7875	27.4375
Z12(6,6)	**12**	56.2375	50.0375	43.8375	37.6375	31.4375
Z13(6,7)	**13**	63.7703	56.7203	49.6703	42.6203	35.5703
Z14(7,7)	**14**	71.3031	63.4031	55.5031	47.6031	39.7031
Z15(7,8)	**15**	78.8984	70.1484	61.3984	52.6484	43.8984

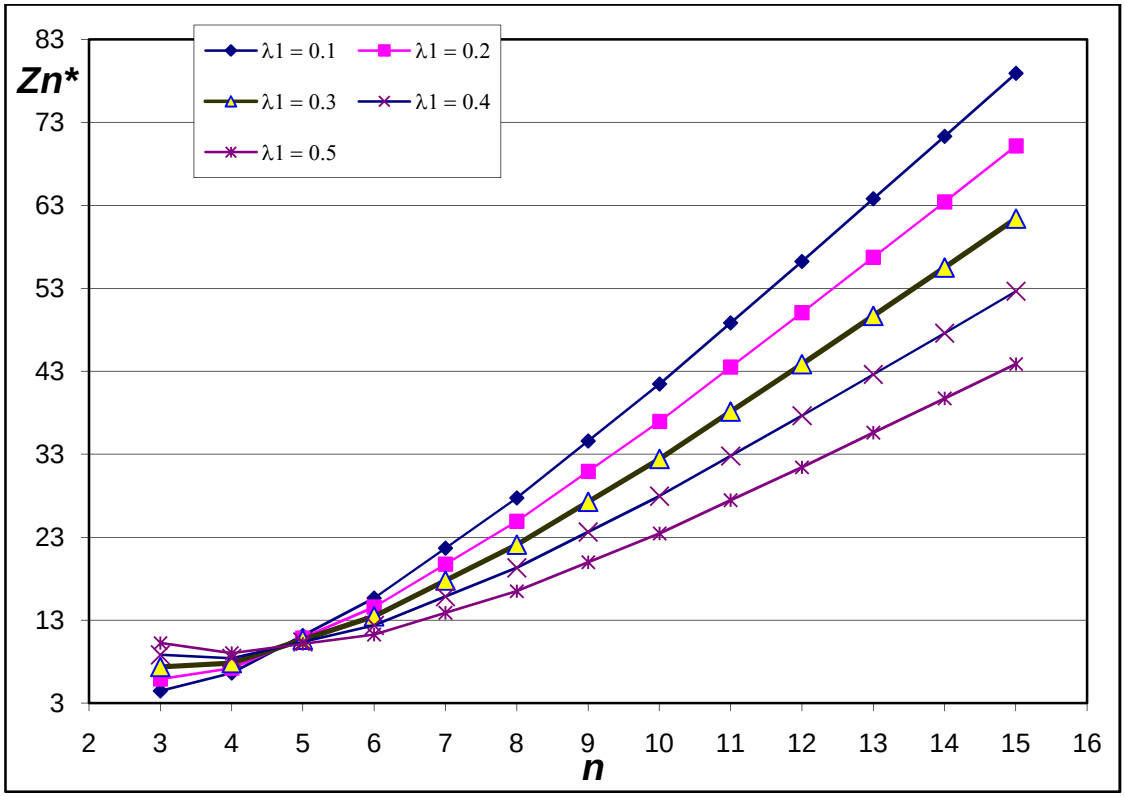


Figure 13: Graph for Package type 1 probability of 0.5 results for 5 values of λ_1

The changes in the probability of selection of package types had different effect on the results as compared to the changes noticed in the previous section. Most penalty cost values were increased as the value of α_1, the probabilty of selection of item 1 increases. The only exception to this trend is $Z_3(1,2)$, the assignment of 3 items. The penalty cost values reduces for $\lambda_1 = 0.3$, 0.4, and 0.5 for tables 15, 16, 17 and 18. The highlighted values in the tables show this trend.

6.5.2 Changes in Processing Times

The processing times for both package types were varied to see the effect on the solutions for the same five values of λ_1 with capacity of 20, and probabilities of 0.2, and 0.8 respectively for the package types. The following tables and graphs resulted from that experiment.

Table 19: Processing times of 16 and 1 results for 5 values of λ_1

prob1=0.2,prob2=0.8,capacity=20 p1=16,p2=1						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	3.2800	6.0800	8.8800	11.6800	14.4800
Z4(2,2)	**4**	3.3600	5.7600	8.1600	10.5600	12.9600
Z5(2,3)	**5**	3.9520	5.9520	7.9520	9.9520	11.9520
Z6(3,3)	**6**	4.5440	6.1440	7.7440	9.3440	10.9440
Z7(3,4)	**7**	5.6352	6.8352	8.0352	9.2352	10.4352
Z8(4,4)	**8**	6.7264	7.5264	8.3264	9.1264	9.9264
Z9(4,5)	**9**	8.2784	8.6784	9.0784	9.4784	9.8784
Z10(5,5)	**10**	9.8304	9.8304	9.8304	9.8304	9.8304
Z11(5,6)	**11**	12.1852	11.7852	11.3852	10.9852	10.5852
Z12(6,6)	**12**	14.5400	13.7400	12.9400	12.1400	11.3400
Z13(6,7)	**13**	17.1963	15.9963	14.7963	13.5963	12.3963
Z14(7,7)	**14**	19.8526	18.2526	16.6526	15.0526	13.4526
Z15(7,8)	**15**	22.7396	20.7396	18.7396	16.7396	14.7396

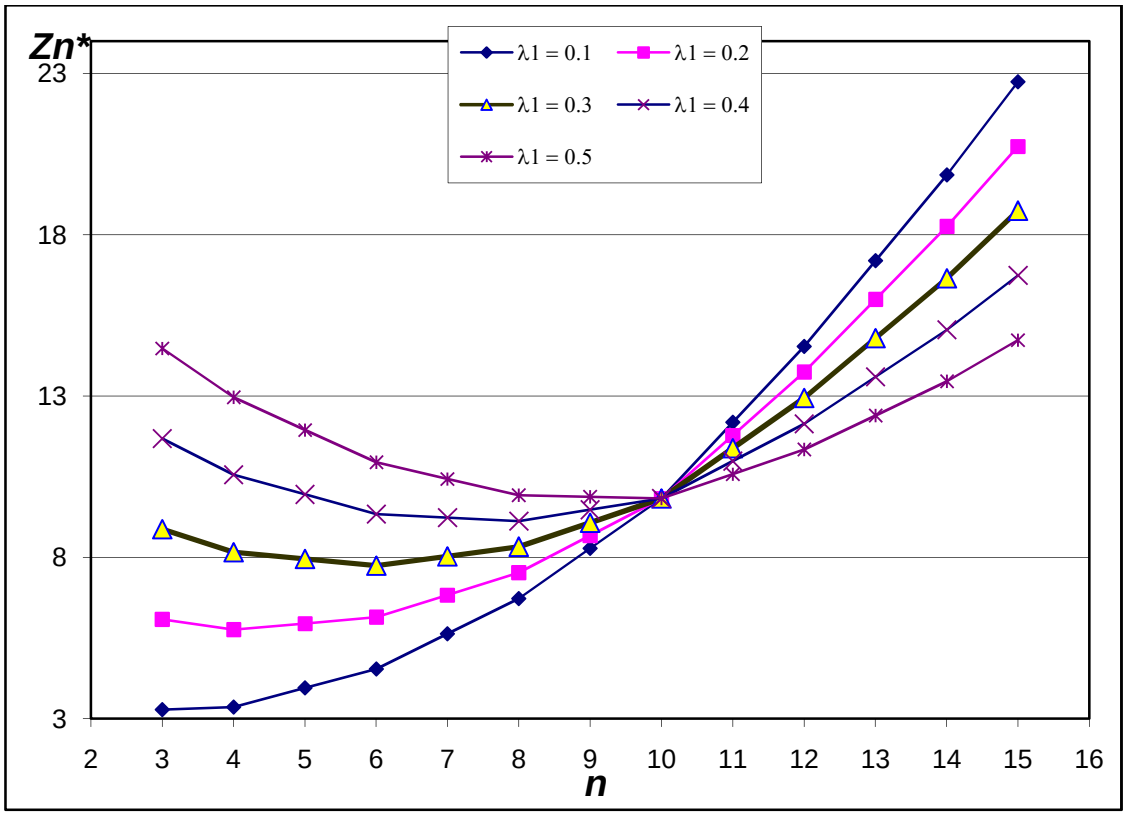


Figure 14: Graph for processing times of 16 and 1 results for 5 values of λ_1

Table 20: Processing times of 14 and 3 results for 5 values of λ_1

prob1=0.2,prob2=0.8,capacity=20 p1=14,p2=3						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	2.7600	5.2000	7.6400	10.0800	12.5200
Z4(2,2)	**4**	2.5600	4.4800	6.4000	8.3200	10.2400
Z5(2,3)	**5**	2.9520	4.3520	5.7520	7.1520	8.5520
Z6(3,3)	**6**	3.3440	4.2240	5.1040	5.9840	6.8640
Z7(3,4)	**7**	5.6688	6.0288	6.3888	6.7488	7.1088
Z8(4,4)	**8**	7.9936	7.8336	7.6736	7.5136	7.3536
Z9(4,5)	**9**	11.0352	10.3552	9.6752	8.9952	8.3152
Z10(5,5)	**10**	14.0768	12.8768	11.6768	10.4768	9.2768
Z11(5,6)	**11**	17.6427	15.9227	14.2027	12.4827	10.7627
Z12(6,6)	**12**	21.2086	18.9686	16.7286	14.4886	12.2486
Z13(6,7)	**13**	25.3643	22.6043	19.8443	17.0843	14.3243
Z14(7,7)	**14**	29.5200	26.2400	22.9600	19.6800	16.4000
Z15(7,8)	**15**	34.2000	30.4000	26.6000	22.8000	19.0000

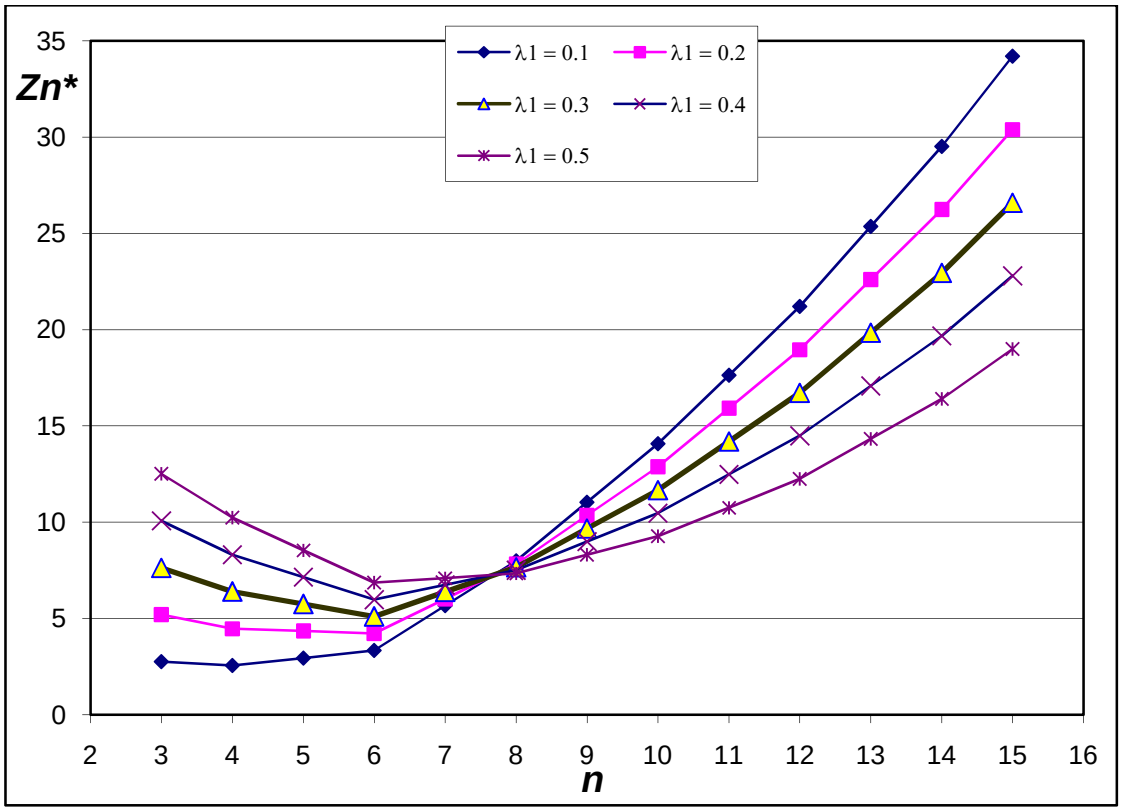


Figure 15: Graph for processing times of 14 and 3 results for 5 values of λ_1

Table 21: Processing times of 12 and 5 results for 5 values of λ_1

prob1=0.2,prob2=0.8,capacity=20 p1=12,p2=5						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	2.2400	4.3200	6.4000	8.4800	10.5600
Z4(2,2)	**4**	1.7600	3.2000	4.6400	6.0800	7.5200
Z5(2,3)	**5**	2.7200	3.5200	4.3200	5.1200	5.9200
Z6(3,3)	**6**	3.6800	3.8400	4.0000	4.1600	4.3200
Z7(3,4)	**7**	6.8800	6.4000	5.9200	5.4400	4.9600
Z8(4,4)	**8**	10.0800	8.9600	7.8400	6.7200	5.6000
Z9(4,5)	**9**	15.8400	14.0800	12.3200	10.5600	8.8000
Z10(5,5)	**10**	21.6000	19.2000	16.8000	14.4000	12.0000
Z11(5,6)	**11**	27.3600	24.3200	21.2800	18.2400	15.2000
Z12(6,6)	**12**	33.1200	29.4400	25.7600	22.0800	18.4000
Z13(6,7)	**13**	38.8800	34.5600	30.2400	25.9200	21.6000
Z14(7,7)	**14**	44.6400	39.6800	34.7200	29.7600	24.8000
Z15(7,8)	**15**	50.4000	44.8000	39.2000	33.6000	28.0000

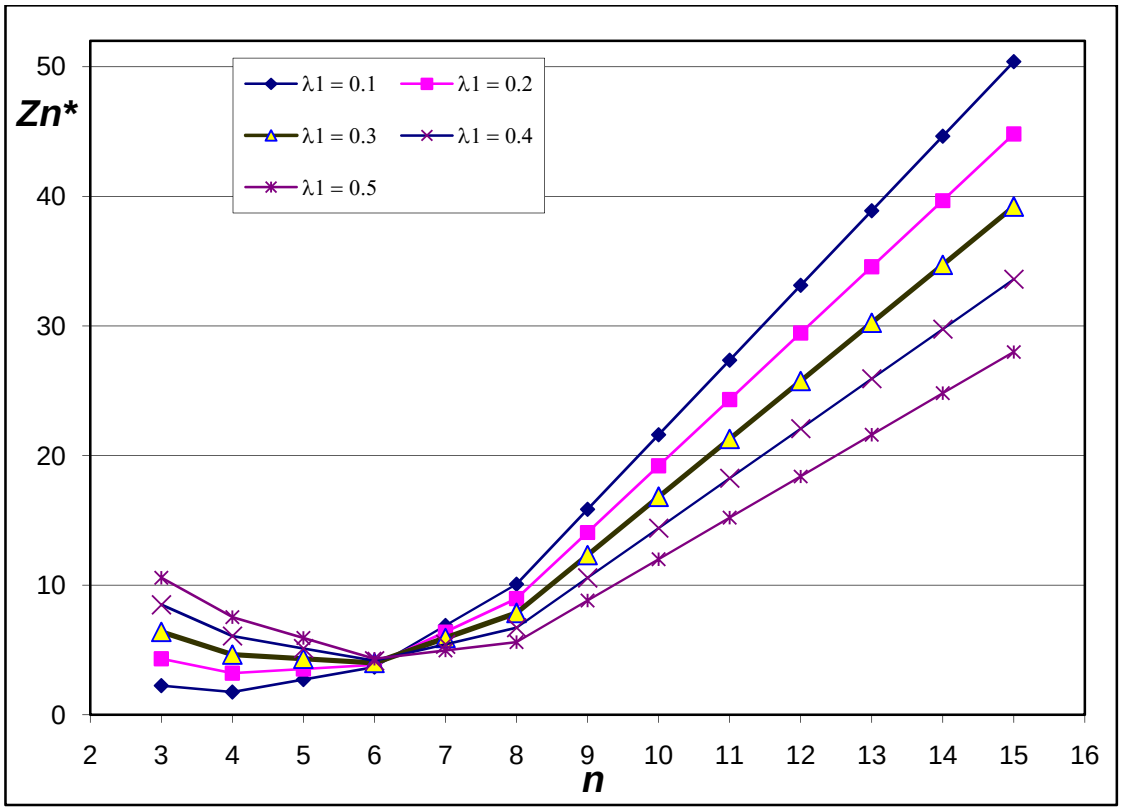


Figure 16: Graph for processing times of 12 and 5 results for 5 values of λ_1

Table 22: Processing times of 10 and 7 results for 5 values of λ_1

prob1=0.2,prob2=0.8,capacity=20 p1=10,p2=7						
	n	**λ1 = 0.1**	**λ1 = 0.2**	**λ1 = 0.3**	**λ1 = 0.4**	**λ1 = 0.5**
Z3(1,2)	**3**	1.7200	3.4400	5.1600	6.8800	8.6000
Z4(2,2)	**4**	0.9600	1.9200	2.8800	3.8400	4.8000
Z5(2,3)	**5**	3.0000	3.2000	3.4000	3.6000	3.8000
Z6(3,3)	**6**	5.0400	4.4800	3.9200	3.3600	2.8000
Z7(3,4)	**7**	11.8800	10.5600	9.2400	7.9200	6.6000
Z8(4,4)	**8**	18.7200	16.6400	14.5600	12.4800	10.4000
Z9(4,5)	**9**	25.5600	22.7200	19.8800	17.0400	14.2000
Z10(5,5)	**10**	32.4000	28.8000	25.2000	21.6000	18.0000
Z11(5,6)	**11**	39.2400	34.8800	30.5200	26.1600	21.8000
Z12(6,6)	**12**	46.0800	40.9600	35.8400	30.7200	25.6000
Z13(6,7)	**13**	52.9200	47.0400	41.1600	35.2800	29.4000
Z14(7,7)	**14**	59.7600	53.1200	46.4800	39.8400	33.2000
Z15(7,8)	**15**	66.6000	59.2000	51.8000	44.4000	37.0000

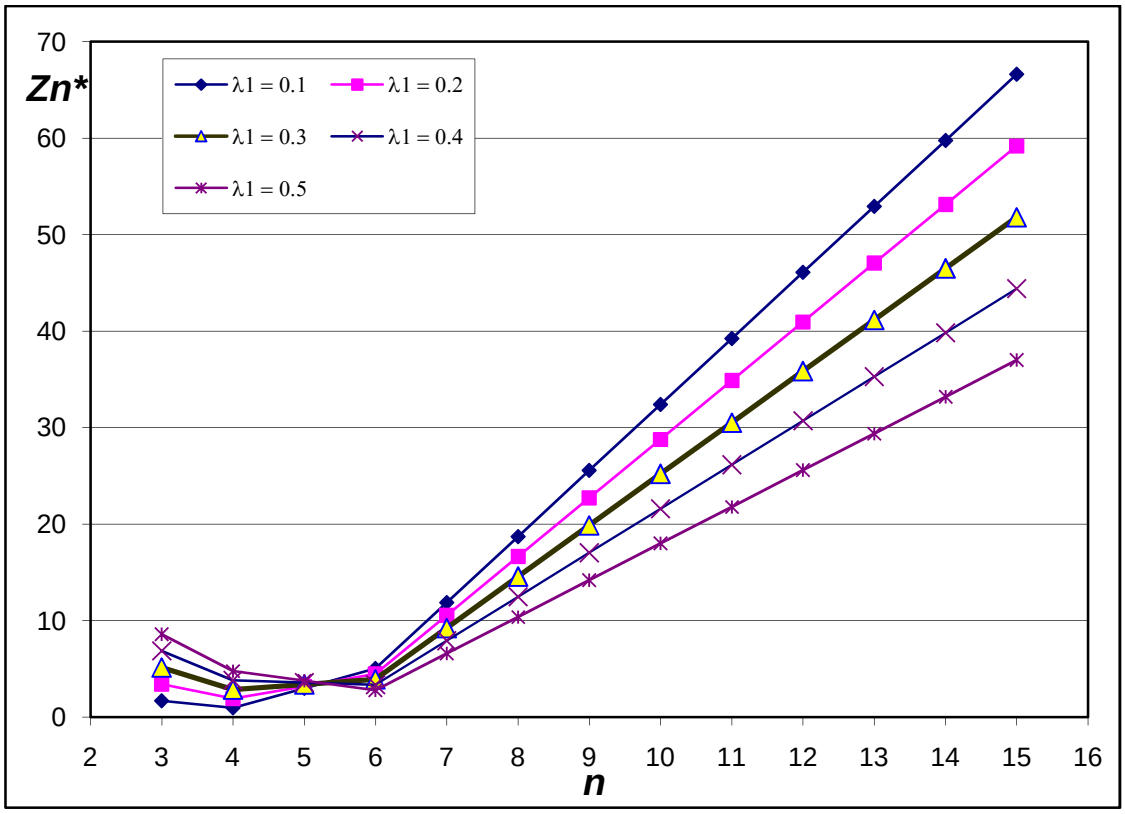


Figure 17: Graph for processing times of 10 and 7 results for 5 values of λ_1

The results obtained for changing the processing times of the package types had decrease in penalty costs for some assignments and increase for some. The assignment of up to 6 items all had reductions in the corresponding values between tables, while the rest of the tables show increasing data values between subsequent tables for selection of 7 to 15 items. This pattern was observed for all data values recorded.

6.6 Further Experimentation

Further experiments were conducted to see which variable has the most significant effect on the penalty cost. Therefore, one variable is changed at a time while

the others were kept constant. The focus was on the processing times and the available resources or capacity. The probabilities were kept constant at 0.2 and 0.8 for the package types 1 and 2 respectively, while $\lambda_1=\lambda_2= 0.5$ for all recorded results.

6.6.1 Changing p_2 with p_1 constant

Processing time for package type 2 was increased while keeping that of type 1 constant, and the processing time of package type 1 was decreased with that of type 2 constant. Five capacity values of 20, 25, 30, 35, and 40 were used. The tables with the graphs on the next pages were recorded.

Table 23: p1=16, p2=1 values for 5 capacities

p1 = 16, p2 = 1					
n	**k = 20**	**k = 25**	**k = 30**	**k = 35**	**k = 40**
Z3(1,2)	14.4800	19.2800	24.0800	29.0000	34.0000
Z4(2,2)	12.9600	17.5600	22.1600	27.0000	32.0000
Z5(2,3)	11.9520	16.2320	20.5120	25.1040	30.0640
Z6(3,3)	10.9440	14.9040	18.8640	23.2080	28.1280
Z7(3,4)	10.4352	14.0112	17.5872	21.5088	26.3328
Z8(4,4)	9.9264	13.1184	16.3104	19.8096	24.5376
Z9(4,5)	9.8784	12.6608	15.4432	18.3792	22.9536
Z10(5,5)	9.8304	12.2032	14.5760	16.9488	21.3696
Z11(5,6)	10.5852	12.1552	14.1184	16.0816	20.0518
Z12(6,6)	11.3400	12.1072	13.6608	15.2144	18.7341
Z13(6,7)	12.3963	12.4295	13.5899	14.7502	17.7194
Z14(7,7)	13.4526	12.7518	13.5189	14.2861	16.7048
Z15(7,8)	14.7396	13.3991	13.7993	14.1994	16.0125

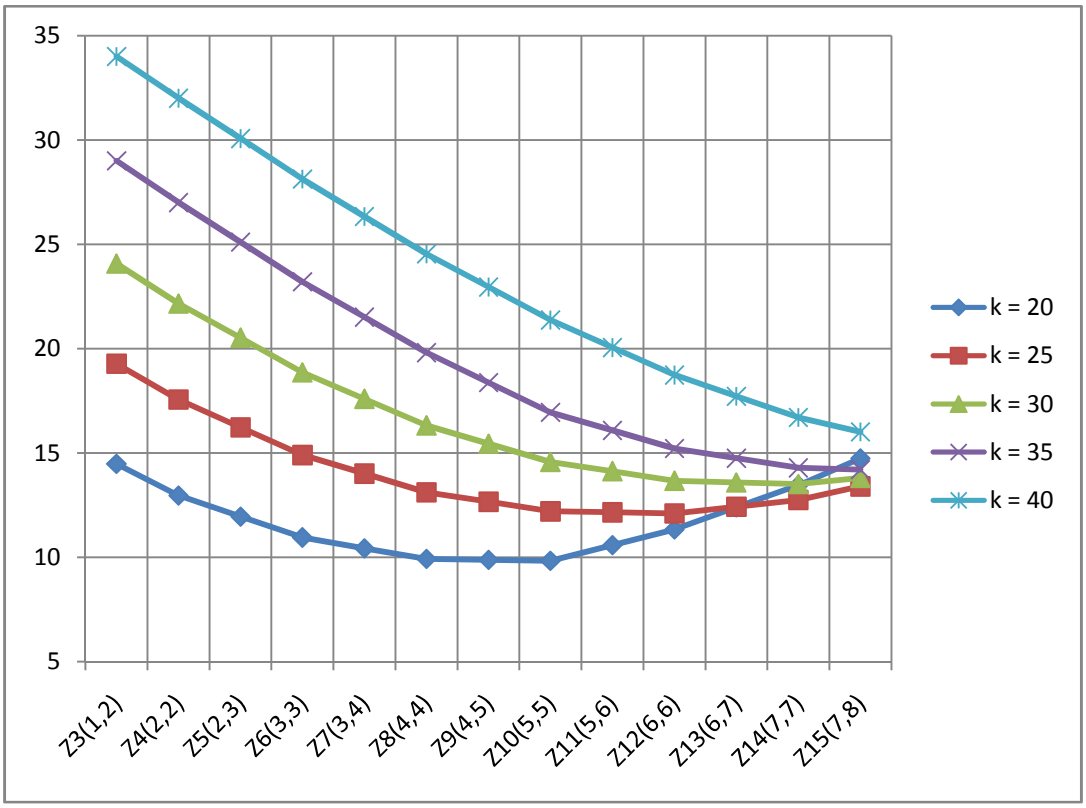


Figure 18: Graphical representation of Table 21

Table 24: p1=16, p2=3 values for 5 capacities

p1 = 16, p2 = 3					
	20	**25**	**30**	**35**	**40**
Z3(1,2)	12.0800	16.8800	21.6800	26.6000	31.6000
Z4(2,2)	9.7600	14.3600	18.9600	23.8000	28.8000
Z5(2,3)	8.9120	12.4240	16.7040	21.1040	26.0640
Z6(3,3)	8.0640	10.4880	14.4480	18.4080	23.3280
Z7(3,4)	8.5088	9.2688	12.8448	16.4208	20.7840
Z8(4,4)	8.9536	8.0496	11.2416	14.4336	18.2400
Z9(4,5)	10.1152	8.8016	10.3552	13.1376	16.2272
Z10(5,5)	11.2768	9.5536	9.4688	11.8416	14.2144
Z11(5,6)	12.9627	10.9118	9.6801	11.2501	13.2133
Z12(6,6)	14.6486	12.2700	9.8915	10.6586	12.2122
Z13(6,7)	16.9243	14.0739	11.4332	10.7323	11.8927
Z14(7,7)	19.2000	15.8777	12.9749	10.8060	11.5732
Z15(7,8)	22.0000	18.0066	14.8941	12.1485	11.8776

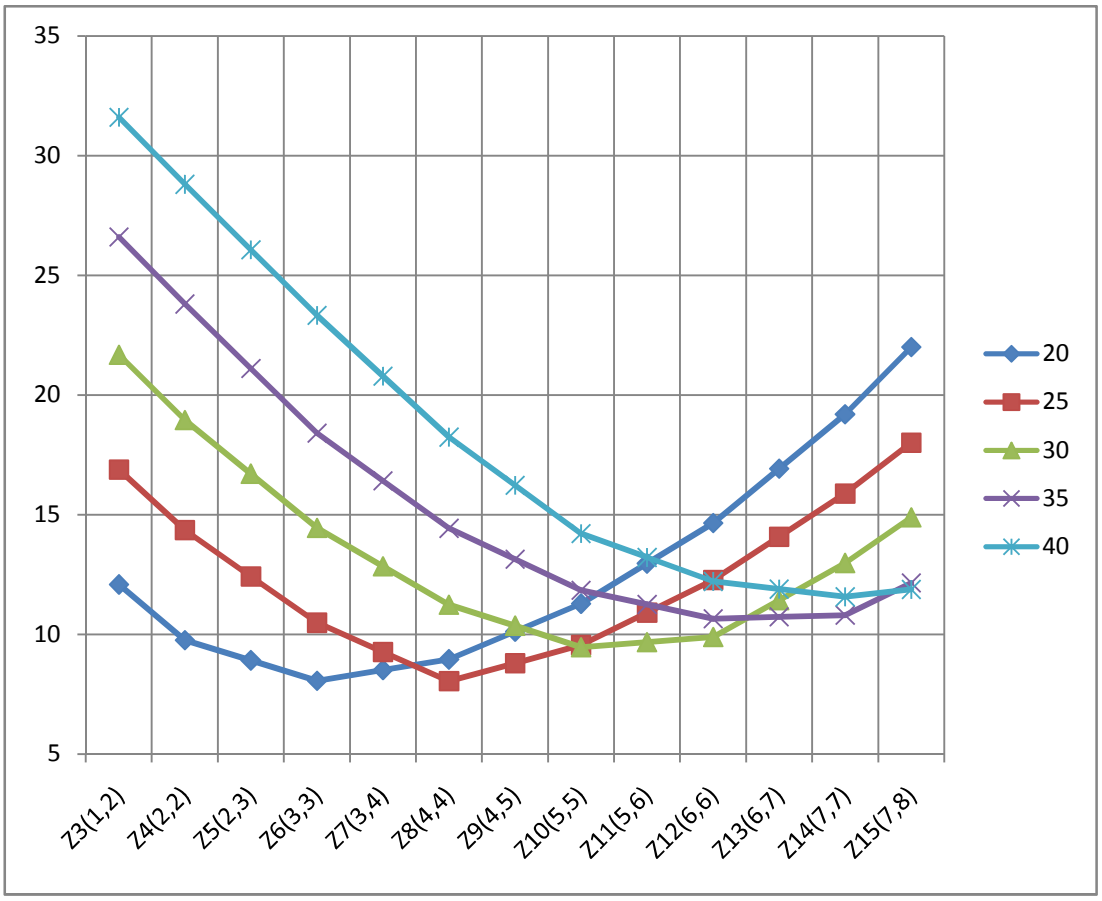


Figure 19: Graphical representation of Table 22

Table 25: p1=16, p2=5 values for 5 capacities

p1 = 16, p2 = 5	20	25	30	35	40
Z3(1,2)	10.0000	14.4800	19.2800	24.2000	29.2000
Z4(2,2)	7.2000	11.1600	15.7600	20.6000	25.6000
Z5(2,3)	6.9600	9.0000	12.8960	17.2960	22.0640
Z6(3,3)	6.7200	6.8400	10.0320	13.9920	18.5280
Z7(3,4)	7.7600	7.3680	8.5120	11.6784	15.5424
Z8(4,4)	8.8000	7.8960	6.9920	9.3648	12.5568
Z9(4,5)	12.4000	9.4480	8.1344	8.4592	10.8320
Z10(5,5)	16.0000	11.0000	9.2768	7.5536	9.1072
Z11(5,6)	19.6000	14.6000	11.2384	9.1875	8.7750
Z12(6,6)	23.2000	18.2000	13.2000	10.8214	8.4429
Z13(6,7)	26.8000	21.8000	16.8000	13.1107	10.4700
Z14(7,7)	30.4000	25.4000	20.4000	15.4000	12.4972
Z15(7,8)	34.0000	29.0000	24.0000	19.0000	15.0486

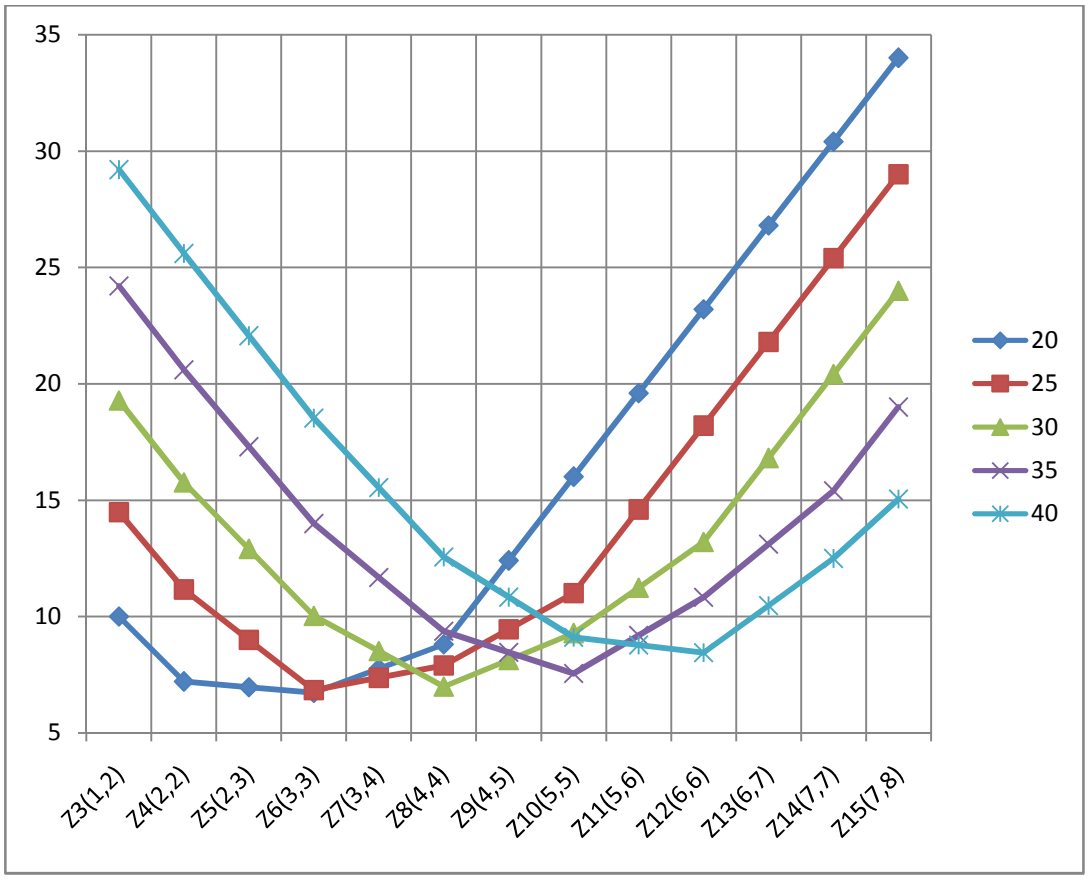


Figure 20: Graphical representation of Table 23

Table 26: p1=16, p2=7 values for 5 capacities

p1 = 16, p2 = 7					
	20	**25**	**30**	**35**	**40**
Z3(1,2)	8.2400	12.0800	16.8800	21.8000	26.8000
Z4(2,2)	5.2800	7.9600	12.5600	17.4000	22.4000
Z5(2,3)	5.8400	6.7280	9.0880	13.4880	18.0640
Z6(3,3)	6.4000	5.4960	5.6160	9.5760	13.7280
Z7(3,4)	10.8000	7.8480	6.2272	7.7552	10.6080
Z8(4,4)	15.2000	10.2000	6.8384	5.9344	7.4880
Z9(4,5)	19.6000	14.6000	10.4192	7.4672	7.3824
Z10(5,5)	24.0000	19.0000	14.0000	9.0000	7.2768
Z11(5,6)	28.4000	23.4000	18.4000	13.4000	10.0384
Z12(6,6)	32.8000	27.8000	22.8000	17.8000	12.8000
Z13(6,7)	37.2000	32.2000	27.2000	22.2000	17.2000
Z14(7,7)	41.6000	36.6000	31.6000	26.6000	21.6000
Z15(7,8)	46.0000	41.0000	36.0000	31.0000	26.0000

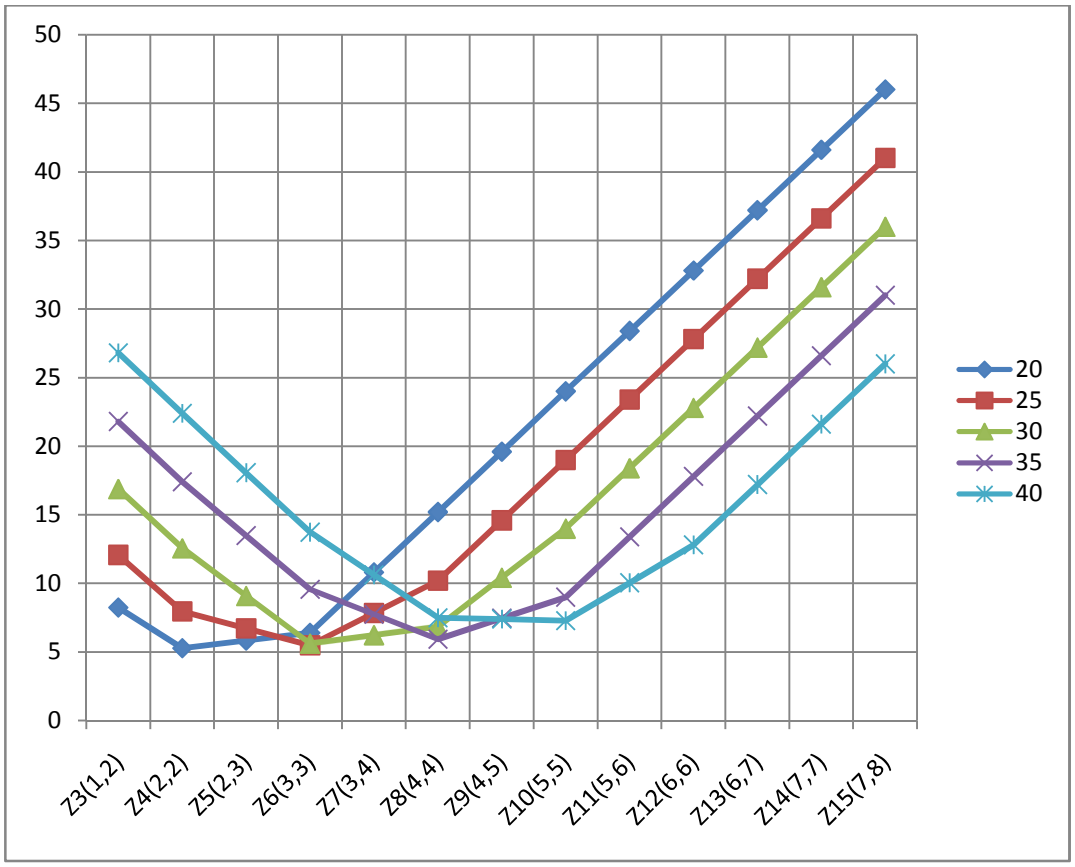


Figure 21: Graphical representation of Table 24

Table 27: p1=16, p2=9 values for 5 capacities

p1 = 16, p2 = 9					
	20	**25**	**30**	**35**	**40**
Z3(1,2)	6.4800	9.6800	14.4800	19.4000	24.4000
Z4(2,2)	3.3600	4.7600	9.3600	14.2000	19.2000
Z5(2,3)	7.2800	5.4800	6.8160	9.6800	14.1600
Z6(3,3)	11.2000	6.2000	4.2720	5.1600	9.1200
Z7(3,4)	16.4000	11.4000	7.9360	5.8800	6.9984
Z8(4,4)	21.6000	16.6000	11.6000	6.6000	4.8768
Z9(4,5)	26.8000	21.8000	16.8000	11.8000	8.4384
Z10(5,5)	32.0000	27.0000	22.0000	17.0000	12.0000
Z11(5,6)	37.2000	32.2000	27.2000	22.2000	17.2000
Z12(6,6)	42.4000	37.4000	32.4000	27.4000	22.4000
Z13(6,7)	47.6000	42.6000	37.6000	32.6000	27.6000
Z14(7,7)	52.8000	47.8000	42.8000	37.8000	32.8000
Z15(7,8)	58.0000	53.0000	48.0000	43.0000	38.0000

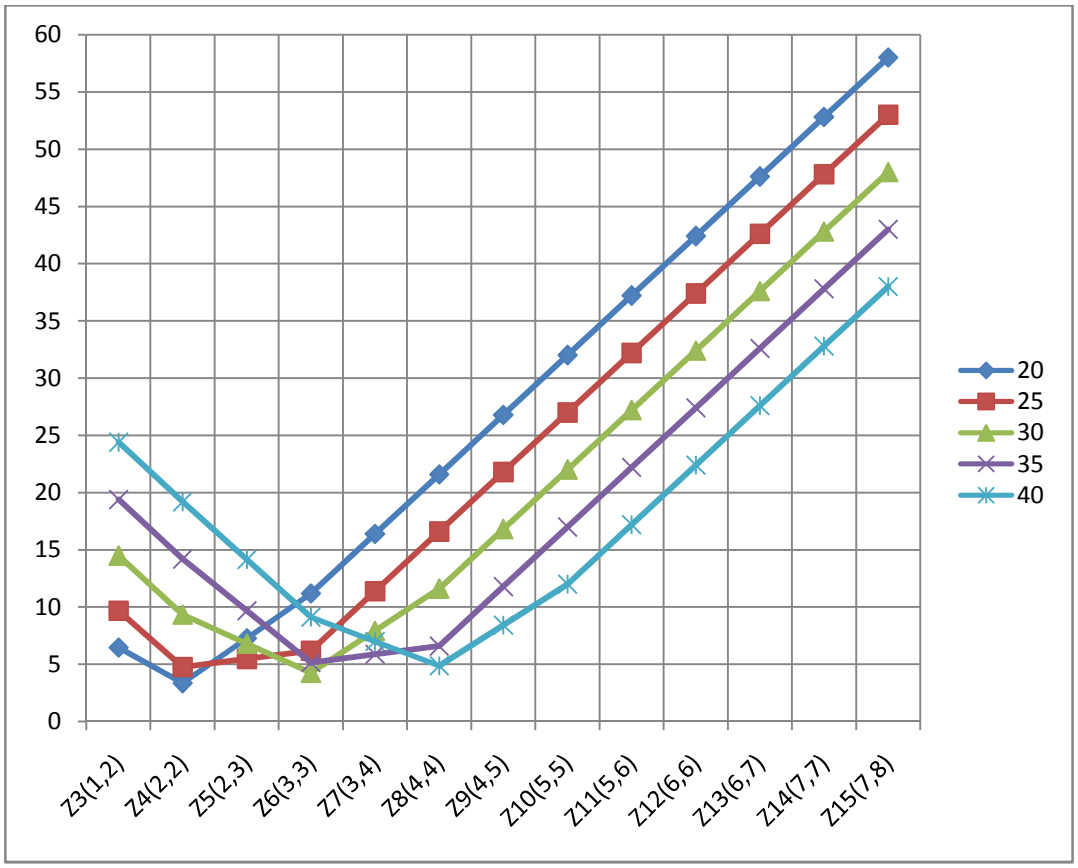


Figure 22: Graphical representation of Table 25

Increasing the processing times of package 2 while keeping that of 1 constant resulted in more increased number of corresponding data values between the tables. For example, the recorded values for capacity of 40 resulted in decrease in all values between table 23 and 24, while 2 vales increased between tables 24 and 25. Five data values increased between tables 25 and 26, and 7 values increased in tables 26 and 27. Generally, a total of 12, 25, 36, and 45 penalty costs increased respectively between the tables. This trend is shown on the tables with the underlined numbers.

6.6.2 Changing p_1 with p_2 constant

Processing time for package type 1 was decreased while keeping that of type 2 constant. Five capacity values of 20, 25, 30, 35, and 40 were used.

The tables with the graphs following were recorded.

Table 28: p1=14, p2=1 values for 5 capacities

p1 = 14, p2 = 1	20	25	30	35	40
Z3(1,2)	14.9200	19.7200	24.6000	29.6000	34.6000
Z4(2,2)	13.4400	18.0400	22.8000	27.8000	32.8000
Z5(2,3)	12.3600	16.6400	21.0960	26.0560	31.0160
Z6(3,3)	11.2800	15.2400	19.3920	24.3120	29.2320
Z7(3,4)	10.6224	14.1984	17.8704	22.6944	27.5184
Z8(4,4)	9.9648	13.1568	16.3488	21.0768	25.8048
Z9(4,5)	9.7168	12.4992	15.2816	19.6512	24.2256
Z10(5,5)	9.4688	11.8416	14.2144	18.2256	22.6464
Z11(5,6)	9.5976	11.5608	13.5240	17.0437	21.2597
Z12(6,6)	9.7265	11.2801	12.8337	15.8618	19.8730
Z13(6,7)	10.1895	11.3499	12.5103	14.9585	18.7239
Z14(7,7)	10.6526	11.4198	12.1869	14.0551	17.5748
Z15(7,8)	11.7396	11.8042	12.2044	13.4486	16.6931

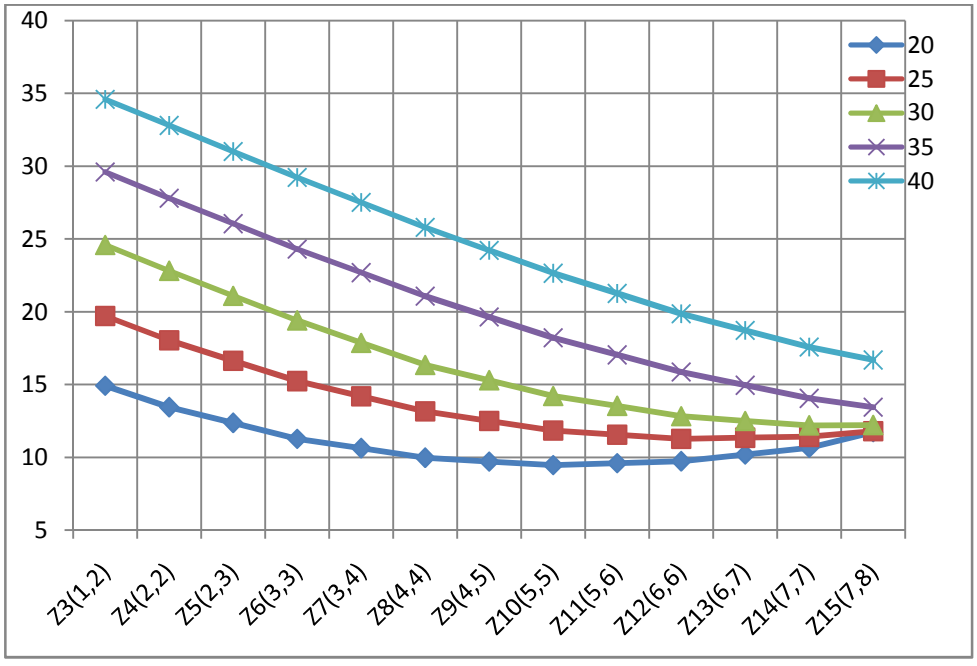


Figure 23: Graphical representation of Table 26

Table 29: p1=12, p2=1 values for 5 capacities

p1 = 12, p2 = 1	20	25	30	35	40
Z3(1,2)	15.3600	20.2000	25.2000	30.2000	35.2000
Z4(2,2)	13.9200	18.6000	23.6000	28.6000	33.6000
Z5(2,3)	12.7680	17.0880	22.0480	27.0080	32.0000
Z6(3,3)	11.6160	15.5760	20.4960	25.4160	30.4000
Z7(3,4)	10.8096	14.3856	19.0560	23.8800	28.8128
Z8(4,4)	10.0032	13.1952	17.6160	22.3440	27.2256
Z9(4,5)	9.5552	12.3376	16.3488	20.9232	25.6768
Z10(5,5)	9.1072	11.4800	15.0816	19.5024	24.1280
Z11(5,6)	9.0033	10.9665	14.0356	18.2516	22.6519
Z12(6,6)	8.8993	10.4529	12.9896	17.0008	21.1758
Z13(6,7)	9.1100	10.2703	12.1975	15.9629	19.8103
Z14(7,7)	9.3206	10.0878	11.4054	14.9251	18.4448
Z15(7,8)	9.8091	10.2093	10.8847	14.1291	17.3736

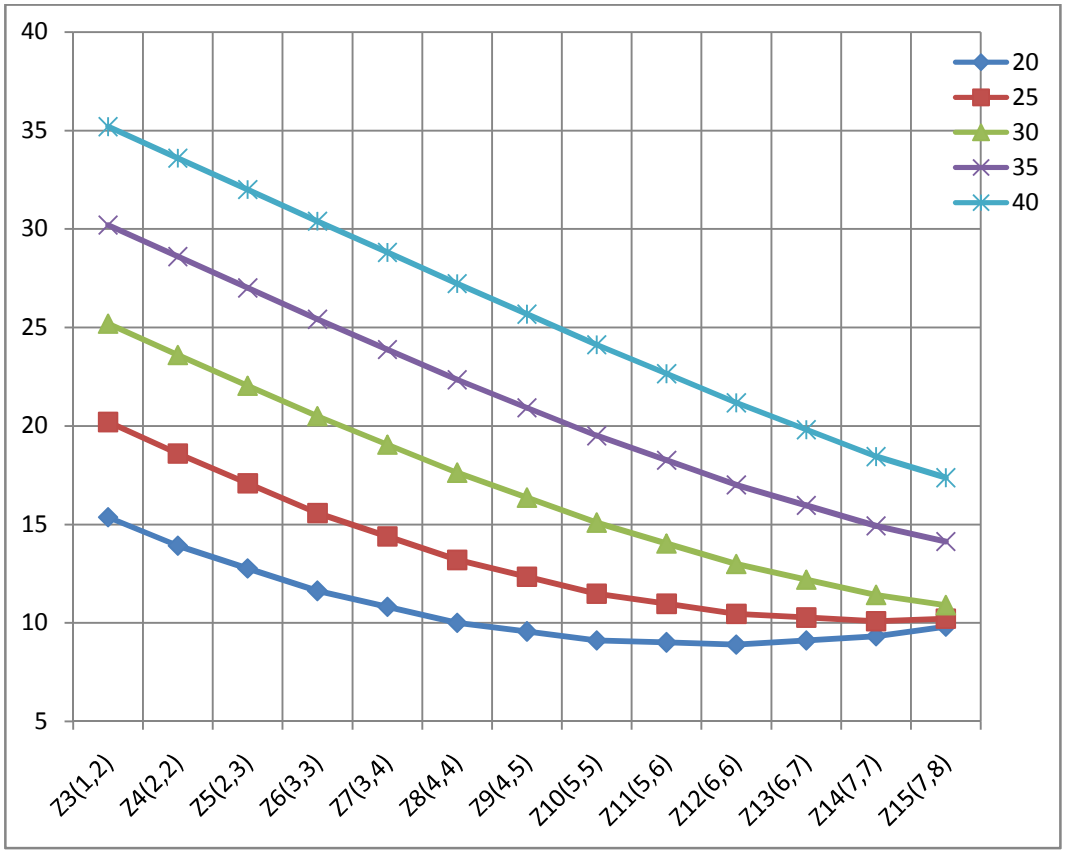


Figure 24: Graphical representation of Table 27

Table 30: p1=10, p2=1 values for 5 capacities

p1 = 10, p2 = 1	20	25	30	35	40
Z3(1,2)	15.8000	20.8000	26.4000	30.8000	35.8000
Z4(2,2)	14.4000	19.4000	25.2000	29.4000	34.4000
Z5(2,3)	13.1760	18.0400	24.0000	28.0000	33.0000
Z6(3,3)	11.9520	16.6800	22.8000	26.6000	31.6000
Z7(3,4)	10.9968	15.4176	21.6032	25.2080	30.2000
Z8(4,4)	10.0416	14.1552	20.4064	23.8160	28.8000
Z9(4,5)	9.3936	13.0464	19.2256	22.4512	27.4096
Z10(5,5)	8.7456	11.9376	18.0448	21.0864	26.0192
Z11(5,6)	8.4089	11.0275	15.2435	19.7769	24.6585
Z12(6,6)	8.0722	10.1173	14.1285	18.4674	23.2978
Z13(6,7)	8.0304	9.4365	13.2020	17.2460	21.9944
Z14(7,7)	7.9886	8.7558	12.2755	16.0245	20.6911
Z15(7,8)	8.2142	8.6144	11.5652	14.9243	19.4762

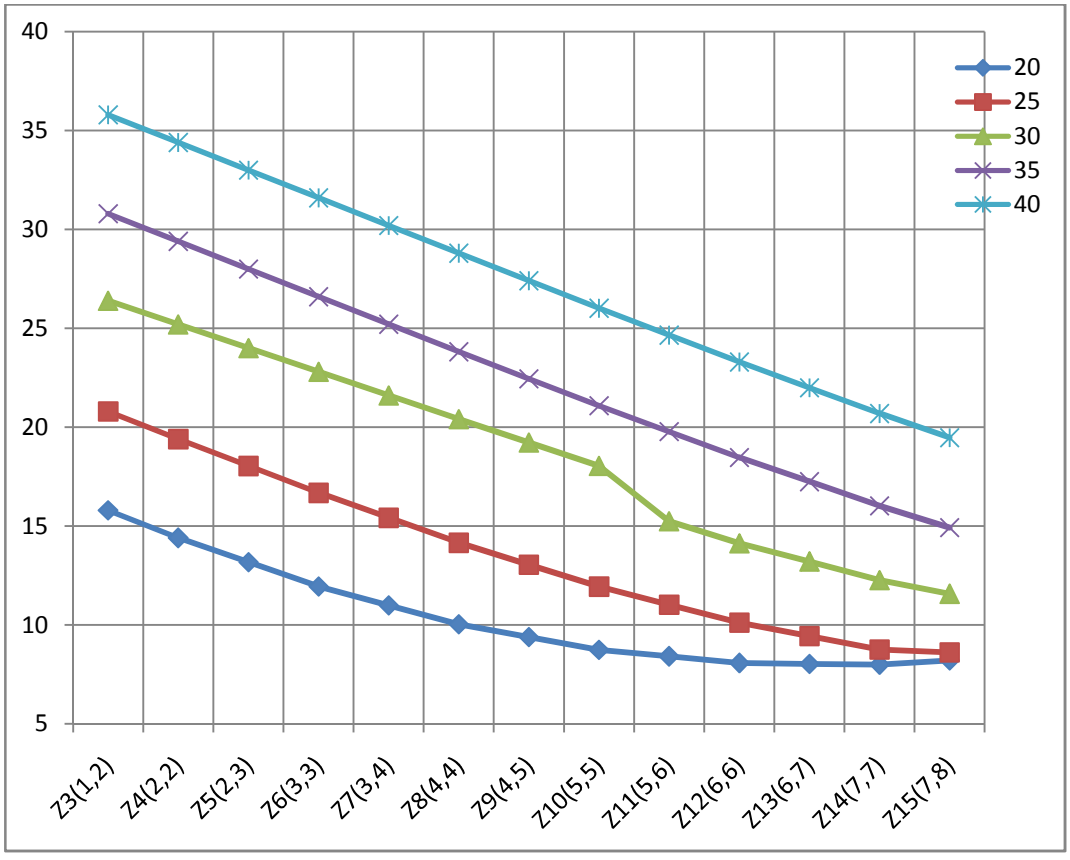


Figure 25: Graphical representation of Table 28

Table 31: p1=8, p2=1 values for 5 capacities

p1 = 8, p2 = 1	20	25	30	35	40
Z3(1,2)	16.4000	21.4000	26.4000	31.4000	36.4000
Z4(2,2)	15.2000	20.2000	25.2000	30.2000	35.2000
Z5(2,3)	14.0320	19.0000	24.0000	29.0000	34.0000
Z6(3,3)	12.8640	17.8000	22.8000	27.8000	32.8000
Z7(3,4)	11.7792	16.6112	21.6032	26.6000	31.6000
Z8(4,4)	10.6944	15.4224	20.4064	25.4000	30.4000
Z9(4,5)	9.7440	14.3184	19.2256	24.2016	29.2000
Z10(5,5)	8.7936	13.2144	18.0448	23.0032	28.0000
Z11(5,6)	8.0193	12.2353	16.9019	21.8116	26.8020
Z12(6,6)	7.2451	11.2563	15.7590	20.6201	25.6041
Z13(6,7)	6.9508	10.4410	14.6816	19.4454	24.4141
Z14(7,7)	6.6566	9.6258	13.6042	18.2708	23.2241
Z15(7,8)	6.6193	9.0013	12.6219	17.1737	22.0525

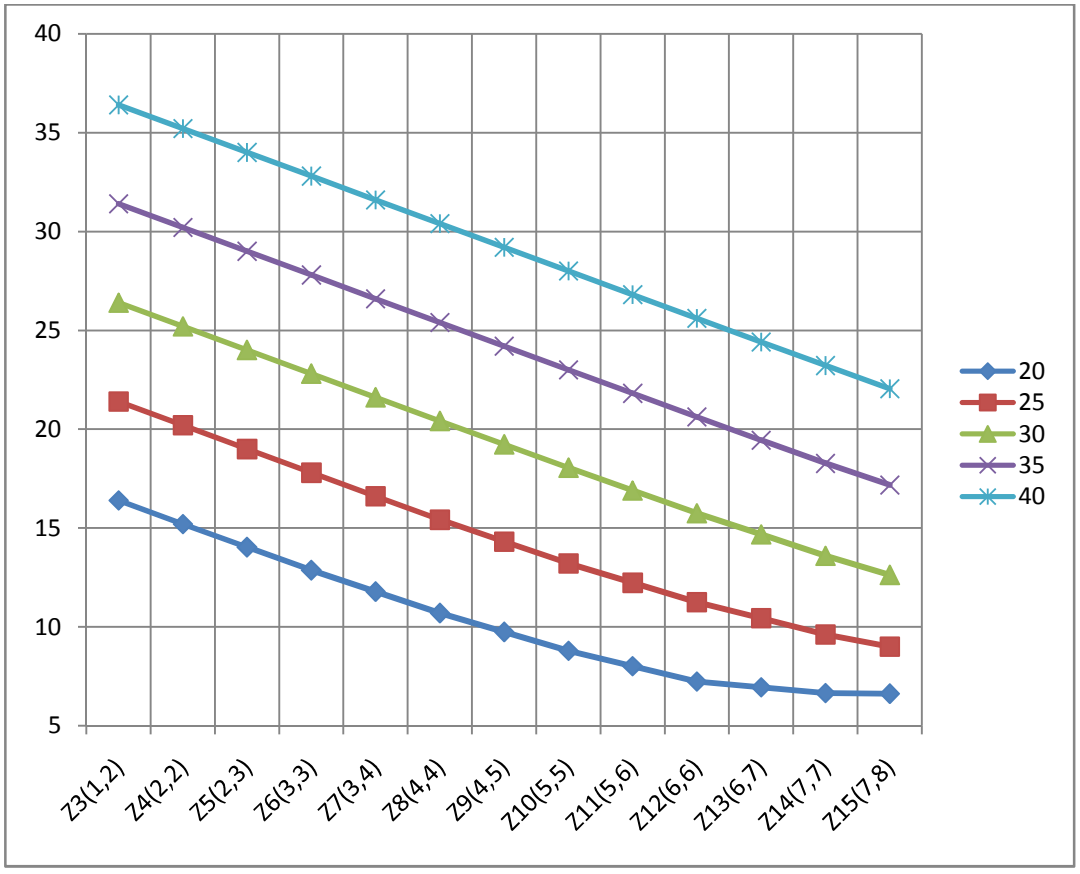


Figure 26: Graphical representation of Table 29

Increasing the processing times of package 1 while keeping that of 2 constant resulted in lesser decreased number of corresponding data values between the tables. For example, the recorded values for capacity of 25 resulted in 7 decreases between table 28 and 29, while 4 vales decreased between tables 29 and 30. Seventeen data values decreased between tables 28 and 29, and 12 values decreased in tables 29 and 30. Generally, a total of 17, 12, and 5 penalty costs decreased respectively between the tables. This trend is shown on the tables with the underlined numbers.

6.7 Results and Analysis

Recall that $Z_n(j, n-j)$ where j = 0, 1, . . , n, refers to the objective function value for each problem. As one would expect, the minimal penalty cost occurred when the packages were equally distributed among the two inspectors for even number of packages. For odd number of packages, minimal penalty cost occurred when one inspector inspects one more package than the others. That is $Z_n(\frac{n}{2}, \frac{n}{2})$ resulted in the minimum penalty cost for even n values and $Z_n(\frac{n-1}{2}, \frac{n+1}{2})$ was always minimal for odd n values. A further look at the 2-inspector problem with a change in the processing time of package type 2 from 1 time units to 3 time units with all other variables held constant yielded the graph below:

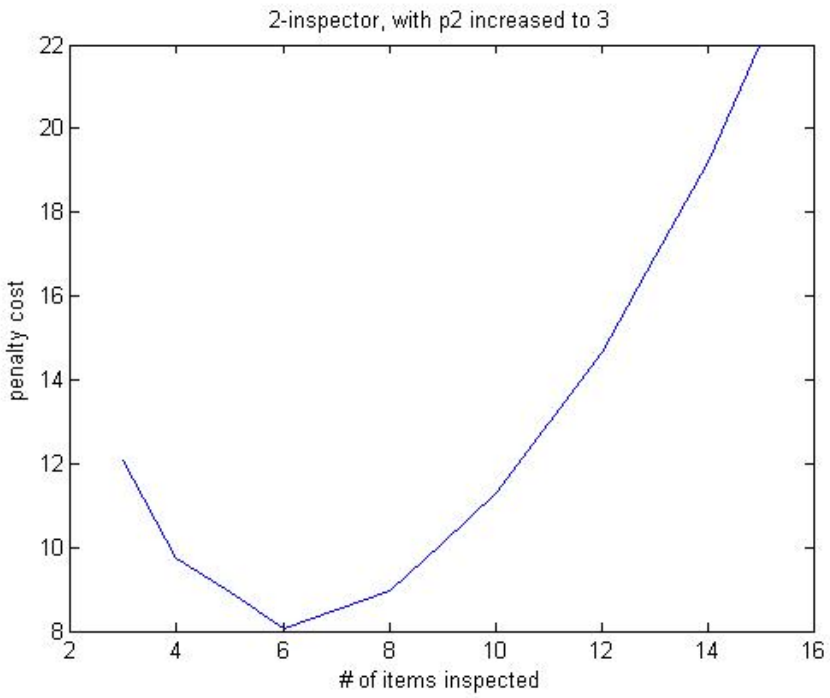


Figure 27: Graph of 2-inspector problem with p_2 increased to 3

The graph above shows a decrease in the optimal objective function value from 9.8304 to 8.0640, with optimal scenario changing from $Z_{10}(5, 5)$ to $Z_6(3, 3)$. This is to be expected since the processing time for both packages is bigger than in the previous problem, meaning fewer packages could be inspected in the same time period.

The variables all had different effects on the solution of the problems. The effects all depend on the problem formulated. Generally speaking penalty costs values will increase faster, if the larger processing time is assigned the bigger probability of selection. The same cannot be confirmed for changes in penalty function, capacity, and processing time. The most noticeable and constant effect is the convex property of all results recorded for all data sets.

Chapter 7

Conclusions and Further Research

7.1 Conclusions

In this research, motivated by Knapsack Problems (KP), we looked at various KP methods, models, and applications.

We developed a new method for solving *MKAR*, multiple knapsack problems with assignment restrictions, a variant of the *MKP* (multiple knapsack problems). Efficient results were obtained by implementing both the developed algorithm, and existing ones. Nine existing assignment procedures with the developed one, *LUCF* – largest unutilized capacity first, were implemented on several generated *KP* (knapsack problems) and the initial feasible solutions for all problems recorded. These results were compared using three measurement yardsticks; the minimum, average, and maximum values returned by each procedure. Three assignment procedures including *LUCF* procedure showed the most promising results in all categories. The *LUCF* algorithm was among the best greedy assignment method for obtaining initial starting feasible solutions for the problems solved, and hence could be implemented in any multiple knapsack problems where an initial feasible solution is required.

Tabu search was employed to improve on the initial feasible results obtained for the *MKAR* by the ten assignment procedures. This was carried out through three major procedures. The main procedure was to get the best feasible solution from all the assignment procedures. The intensification procedure has three parts to obtain the best

solution possible, and the diversification procedure was to ensure that other solutions are explored. The implementation of tabu search procedure led to improvement on the *MKAR*'s initial feasible solutions every time. The only exceptions occur when maximum capacity utilization was the initial solution obtained by any of the tens procedures employed.

Also, motivated by airport security package inspection, Stochastic Knapsack Problem with Penalty Cost (SKPPC), a variant of SKP (stochastic knapsack problem), was formulated and studied. Formulations were created for both the 1-processor, and *m*-processor set-ups. The problem involves the selection of two-item types, and two kinds of problem were investigated. The first was to find the maximum number of packages that would be assigned within a time period. The second problem looked at involves maximizing both the number of packages assigned and the minimum number of inspectors that would be required. The main objective for both problems was to minimize the expected penalty costs. Penalty cost is incurred for both under-utilization and over-utilization of resources. The variables of the problems were varied to see their effects on the solutions obtained. These include changes in the processing times of the package types, the probabilities of selection of the processing times of package types, the penalty cost function, and the resources available to the inspectors. All recorded data were also graphed for visual presentation and analysis. The objective to minimize the total expected penalty cost was easily achieved for all problems solved. The research also includes a proof of the convexity property exhibited by this particular problem.

7.2 Further Research

Further research could be done on MKAR in terms of finding a single algorithm to solve the problem to near optimality as quickly as possible, rather than using an improvement procedure on an initial feasible solution.

The number of item types for the SKPPC could be increased to accommodate various kinds of problems of that nature. The actual penalty function, λ, should be assigned a real value to see its effect on results obtained. A complete enumeration of a problem should involve both the penalty cost from under-utilization, over-utilization, and inventory cost on unassigned items.

Bibliography

[1] Ahrens, J.H. and Finke, G. (1975), Merging and Sorting Applied to the 0-1 Knapsack problems, *Operations Research*, 23, pp. 1099-1109.

[2] Balas, E. and Zemel, E. (1980), An algorithm for large 0-1 Knapsack Problems, *Operations Research*, 28, pp. 1130-1154.

[3] Bellman, R. E. (1957), Dynamic Programming, Princeton University Press, Princeton, NJ.

[4] Caprara, A., Pisinger, D. and Toth, P. (2003), Exact solution of the Quadratic knapsack problem, *INFORMS Journal on Computing*, Vol. 11, pp. 125 - 137.

[5] Cohen, R., Katzir, L. and Raz, D. (2006), An efficient approximation for the Generalized Assignment problem, *Information Processing Letters*, 100, pp. 162 -166.

[6] Chu, P. C. and Beasley, J. E. (1998). A Genetic Algorithm for the Multidimensional Knapsack Problem, *Journal of Heuristics*, 4, pp. 63 - 86.

[7] Dantzig, G.B. (1957), Discrete Variable Problems, *Operations Research*, 5, pp. 266 - 277.

[8] Dawande, M., Kalagnanam, J., Keskinonack, R., Ravi, R. and Salman, F. S. (2000), Approximation Algorithm for the multiple knapsack problems with Assignment Restrictions, *Journal of Combinatorial Optimization*, 4, pp. 171-186.

[9] Dudzinski, K. and Walukiewicz, S. (1987), Exact methods for the Knapsack Problem and its generalizations, *European Journal of Operational Research*, Vol. 28, Issue 1, pp. 3 - 21.

[10] Fayard, D. and Plateau, G. (1977), Reduction Algorithm for single and multiple constraints 0-1 Linear Programming Problems, *Conference on Methods of Mathematical Programming*, Zakopane, Poland.

[11] Glover, F. (1965), A Multiphase Dual Algorithm for the Zero-One Integer Programming Problem. *Operations Research*, 13(6), pp. 879 - 919.

[12] Glover, F. (1986), Future paths for integer programming and links to artificial intelligence, *Computers and Operations Research*, Vol. 1, pp. 549 - 553.

[13] Glover, F. and Kochenberger, G. A. (1996), Critical Event Tabu Search for Multidimensional Knapsack Problems. In Osman, I. H. and Kelly, J. P. (eds), *Meta-Heuristics: Theory and Applications*. Kluwer Academic Publishers, pp. 407- 427.

[14] Glover, F. (1990), Tabu search – Part II. Operations Research Society of America, *Journal on Computing*, Vol. 2, No. 1, pp. 4-32.

[15] Glover, F. and Laguna, M. (1993), Tabu search. In C. R. Reeves (Ed.), *Modern Heuristic Techniques for Combinatorial Problems*, John Wiley & Sons, Inc., New York, pp. 70-150.

[16] Glover and Lokketangen, A. and Glover, F. (1998), Solving Zero-One Mixed Integer Programming Problems using Tabu Search, *European Journal Operational Research*, Vol. 106, Issues 2 - 3, pp. 624 - 658.

[17] Hanafi, S. and Freville, A. (1998), An Efficient Tabu Search Approach for the 0–1 Multidimensional Knapsack Problem, *European Journal Operational Research*, Vol. 106, Issues 2 - 3, pp. 659 - 675.

[18] Hifi, M., Michrafy, M. and Sbihi, A. (2004), Heuristic Algorithms for the Multiple-Choice Multidimensional Knapsack Problem, *Journal of the Operational Research Society*, 55, pp. 1323 - 1332.

[19] Hirschberg, D.S. and Wong, C.K. (1976), A Polynomial-time algorithm for the knapsack problem with two variables, *Journal of ACM*, 23, pp. 147 - 154.

[20] Horowitz, E. and Sahni, S. (1974), Computing partitions with applications to the Knapsack Problem, *Journal of ACM*, 21, pp. 277 - 292.

[21] Hung, M.S. and Fisk, J.C. (1978), An algorithm for 0-1 Multiple Knapsack Problems, *Naval Research Logistics, Quarterly*, 25, pp. 571 - 579.

[22] Keller, H., Pferschy, U. and Pisinger, D. (2004), *Knapsack Problems*. Springer.

[23] Kleywegt, A. J. and Papastavrou, J. D. (1996), The Dynamic and Stochastic Knapsack Problem, *Operations Research*, Vol. 46, No. 1, pp. 17 - 35.

[24] Kleywegt, A. J. and Papastavrou, J. D. (2001), The Dynamic and Stochastic Knapsack Problem with Random Sized Items, *Operations Research*, Vol. 49, No. 1, pp. 26 - 41.

[25] Knox, J. (1989), *The Application of Tabu Search to the Symmetric Traveling Salesman Problem*, Ph.D. thesis, Graduate School of Business, University of Colorado, Boulder.

[26] Lodi, A., Martello, S. and Monaci, M. (2002), Two-dimensional packing problem: a survey, *European Journal of Operational Research*, Vol. 141, pp. 241 - 252.

[27] Martello, S. and Toth, P. (1977), An upper bound for the Zero-One Knapsack Problem and a branch and bound Algorithm, *European Journal of Operational Research,* Vol. 1, Issue 3, pp. 169 - 175.

[28] Martello, S. and Toth, P. (1981), A bound and bound Algorithm for the Zero-One Multiple Knapsack Problem, *Discrete Applied Mathematics*, 3, pp. 275 - 288.

[29] Martello, S. and Toth, P. (1988), A new algorithm for the 0-1 Knapsack Problem, *Management Science*, 34, pp. 633 - 644.

[30] Martello, S. and Toth, P. (1990), Knapsack Problems: *Algorithms and Computer Implementations*, John Wiley & Sons.

[31] Papastavrou, J. D., Rajagopalan, S., and Kleywegt, A. J. (1996), The Dynamic and Stochastic Knapsack Problem with Deadlines, *Management Science*, Vol. 42, No. 12, pp. 1706 - 1718.

[32] Pinedo, M. (1995), Scheduling – Theory, Algorithms and Systems, Prentice Hall, Englewood Cliffs, New Jersey.

[33] Pisinger, D. (1995), A minimal Algorithm for the Multiple-Choice Knapsack problem, *European Journal of Operational Research*, Vol. 83, Issue 2, pp. 394 - 410.

[34] Pisinger, D. (1995), *Algorithms for Knapsack Problems*, PhD thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark.

[35] Pisinger, D. (1999), An exact Algorithm for large Multiple Knapsack Problems, *European Journal of Operational Research*, Vol. 114, pp. 528 - 541.

[36] Pisinger, D. (2000), A Minimal Algorithm for the Bounded Knapsack Problem, *INFORMS Journal on Computing*, Vol. 12, Issue 1, pp. 75 - 82.

[37] Prabandari, A. (1998), *A Tabu-Search-Based Algorithm for unconstrained optimization*, MS thesis, School of Industrial Engineering, University of Oklahoma, Norman, OK.

[38] Premkumar, R. (1998), *Facility Layout optimization using Tabu Search*, MS thesis, School of Industrial Engineering, University of Oklahoma, Norman, OK.

[39] Ross, K. W. and Tsang, D. H. K. (1989), The Stochastic Knapsack Problem, *IEEE Transactions On Communications*, Vol. 37, No. 7, pp. 740 - 747.

[40] Skorin-Kapov, J. (1990), Tabu Search Applied to the Quadratic Assignment Problem, *Journal on Computing*, Vol. 2, No. 1, pp. 33 - 45.

[41] Soma, N.Y. and Toth, P. (2002), An exact algorithm for the subset-sum problem, *European Journal of Operational Research*, Vol. 136, pp. 57 - 66.

[42] Xavier, G. and Arnaud, F. (2000), Tabu Search Based Procedure for Solving the 0-1 MultiObjective Knapsack Problem: The Two Objectives Case, *Journal of Heuristics*, Vol. 6, pp. 361 - 383.

Appendix A

MATLAB Code for the Assignment Algorithms

```
clear

% # of items and knapsacks
n = input ('          enter total number of packagess: ');
disp('  ')
m = input ('            enter number of inspectors: ');
disp('  ')
r = input ('     enter number of replications/run: ');
knaprt = [];

knaprs = zeros (r,10); % storage for knapsack residual space
cap = zeros (r,m); % storage for knapsack capacity
cond = zeros (1,r); % storage for conditions
knaprsum = zeros (1,10); % storage for sum of knapsack unutilized
space


% while r > 0

% generating item weights
a = ceil(random('unif', 10, 100, n, 1));
w = a;
w = sort (w);  % item weights sorted in ascending order

% generating item processing times probabilities
p = vpa((random('unif', 0.1, 0.8, n, 1)),1);

% f = item weights
% f = -1 * w;

a1 = 0.4 * (sum(w)/m)
a2 = 0.6 * (sum(w)/m)

% generating similar knapsack capacities
a = (random('unif', a1, a2, m-1, 1))
a (m, 1) = 0.5 * sum(w) - sum (a)
c = ceil(a)
c= sort (c)  % knapsack capacities sorted in ascending order

cap (r,:)=c'

if max (w) <= min (c) & min (c) >= min (w) & sum (w) > max (c);
   cond (1,r) = 1
end

% b = capacities
% b = c;

% generating A
A = [];
for i = 1 : m
    A = [A ; w'];
end
```

```
% storing the items sorted in ascending order
item1 = zeros(n,2);
for i = 1 : n
    item1 ( i, 1) = i;
    item1 ( i, 2) = w ( i, 1);
end
item1; % items sorted in ascending order

% adding knapsack restrictions to items sorted in ascending order
d1 = [item1 b];

w = w(n:-1:1); % item weights sorted in descending order

% storing the items sorted in descending order
item2 = zeros(n,2);
for i = 1 : n
    item2 ( i, 1) = i;
    item2 ( i, 2) = w ( i, 1);
end
item2; % items sorted in descending order

% adding knapsack restrictions to items sorted in descending order
d2 = [item2 b];

% storing the knapsacks sorted in ascending order
knapc1 = zeros(m,2);
for i = 1 : m
    knapc1 ( i, 1) = i;
    knapc1 ( i, 2) = c ( i, 1);
end
knapc1; % knapsacks sorted in ascending order

c = c(m:-1:1);  % knapsack capacities sorted in descending order
% storing the knapsacks sorted in descending order
knapc2 = zeros(m,2);
for i = 1 : m
    knapc2 ( i, 1) = i;
    knapc2 ( i, 2) = c ( i, 1);
end
knapc2; % knapsacks sorted in descending order

% residual capacities storage
knapr = zeros (10,m);


% MODEL 1

% storage location for knapsack assignment
aknap1 = [zeros(n ,1) item1 zeros(n ,1)];
knap1 = knapc1;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
```

```
        if item1 ( i, 2) <= knap1 ( j, 2)
            aknap1 ( i, 1) = knap1 ( j, 1);
            knap1 ( j, 2) = knap1 ( j, 2) - aknap1 ( i, 3);
            aknap1 ( i, 4) = knap1 ( j, 2);
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end

aknap1;

for i = 1 : m
    knapr (1, i) = knap1 (i, 2);
end
aknap1 % model 1 knapsack assignment

% MODEL 2

% storage location for knapsack assignment
aknap2 = [zeros(n ,1) item1 zeros(n ,1)];
knap2 = knapc1;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if item1 ( i, 2) <= knap2 ( j, 2)
            aknap2 ( i, 1) = knap2 ( j, 1);
            knap2 ( j, 2) = knap2 ( j, 2) - aknap2 ( i, 3);
            aknap2 ( i, 4) = knap2 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end

aknap2;

for i = 1 : m
    knapr (2, i) = knap2 (i, 2);
end
aknap2 % model 2 knapsack assignment
```

```
% MODEL 3

% storage location for knapsack assignment
aknap3 = [zeros(n ,1) item1 zeros(n ,1)];
knap3 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if item1 ( i, 2) <= knap3 ( j, 2)
            aknap3 ( i, 1) = knap3 ( j, 1);
            knap3 ( j, 2) = knap3 ( j, 2) - aknap3 ( i, 3);
            aknap3 ( i, 4) = knap3 ( j, 2);
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap3;

for i = 1 : m
    knapr (3, i) = knap3 ((m+1)-i, 2);
end
aknap3 % model 3 knapsack assignment


% MODEL 4

% storage location for knapsack assignment
aknap4 = [zeros(n ,1) item1 zeros(n ,1)];
knap4 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if item1 ( i, 2) <= knap4 ( j, 2)
            aknap4 ( i, 1) = knap4 ( j, 1);
            knap4 ( j, 2) = knap4 ( j, 2) - aknap4 ( i, 3);
            aknap4 ( i, 4) = knap4 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
```

```
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap4;

for i = 1 : m
    knapr (4, i) = knap4 ((m+1)-i, 2);
end
aknap4 % model 4 knapsack assignment


% MODEL 5

% storage location for knapsack assignment
aknap5 = [zeros(n ,1) item2 zeros(n ,1)];
knap5 = knapc1;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if item2 ( i, 2) <= knap5 ( j, 2)
            aknap5 ( i, 1) = knap5 ( j, 1);
            knap5 ( j, 2) = knap5 ( j, 2) - aknap5 ( i, 3);
            aknap5 ( i, 4) = knap5 ( j, 2);
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end

aknap5;

for i = 1 : m
    knapr (5, i) = knap5 (i, 2);
end
aknap5 % model 5 knapsack assignment


% MODEL 6

% storage location for knapsack assignment
aknap6 = [zeros(n ,1) item2 zeros(n ,1)];
knap6 = knapc1;
```

```
% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if item2 ( i, 2) <= knap6 ( j, 2)
            aknap6 ( i, 1) = knap6 ( j, 1);
            knap6 ( j, 2) = knap6 ( j, 2) - aknap6 ( i, 3);
            aknap6 ( i, 4) = knap6 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap6;

for i = 1 : m
    knapr (6, i) = knap6 (i, 2);
end
aknap6 % model 6 knapsack assignment


% MODEL 7

% storage location for knapsack assignment
aknap7 = [zeros(n ,1) item2 zeros(n ,1)];
knap7 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if item2 ( i, 2) <= knap7 ( j, 2)
            aknap7 ( i, 1) = knap7 ( j, 1);
            knap7 ( j, 2) = knap7 ( j, 2) - aknap7 ( i, 3);
            aknap7 ( i, 4) = knap7 ( j, 2);
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end
```

```
aknap7;

for i = 1 : m
    knapr (7, i) = knap7 ((m+1)-i, 2);
end
aknap7 % model 7 knapsack assignment


% MODEL 8

% storage location for knapsack assignment
aknap8 = [zeros(n ,1) item2 zeros(n ,1)];
knap8 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if item2 ( i, 2) <= knap8 ( j, 2)
            aknap8 ( i, 1) = knap8 ( j, 1);
            knap8 ( j, 2) = knap8 ( j, 2) - aknap8 ( i, 3);
            aknap8 ( i, 4) = knap8 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap8;

for i = 1 : m
    knapr (8, i) = knap8 ((m+1)-i, 2);
end
aknap8 % model 8 knapsack assignment


% MODEL 9

% storage location for knapsack assignment
aknap9 = [zeros(n ,1) item1 zeros(n ,1)];
knap9 = knapc2;

% assiginment of items to knapsacks
for i = 1: n
    knap9 = sortrows(knap9,[2]);
    for j = m:-1:1
        if item1 ( i, 2) <= knap9 ( j, 2)
            aknap9 ( i, 1) = knap9 ( j, 1);
            knap9 ( j, 2) = knap9 ( j, 2) - aknap9 ( i, 3);
```

```
            aknap9 ( i, 4) = knap9 ( j, 2);
            break
        elseif j > 1
            continue
        end
    end
end

aknap9;

for i = 1 : m
    knapr (9, i) = knap9 ((m+1)-i, 2);
end
aknap9 % model 9 knapsack assignment


% MODEL 10

% storage location for knapsack assignment
aknap10 = [zeros(n ,1) item2 zeros(n ,1)];
knap10 = knapc2;

% assignment of items to knapsacks
for i = 1: n
    knap10 = sortrows(knap10,[2]);
    for j = m:-1:1
        if item2 ( i, 2) <= knap10 ( j, 2)
            aknap10 ( i, 1) = knap10 ( j, 1);
            knap10 ( j, 2) = knap10 ( j, 2) - aknap10 ( i, 3);
            aknap10 ( i, 4) = knap10 ( j, 2);
            break
        elseif j > 1
            continue
        end
    end
end

aknap10;

for i = 1 : m
    knapr (10, i) = knap10 ((m+1)-i, 2);
end
aknap10 % model 10 knapsack assignment


% Knapsack residual capacities for all models
knapr

knaprs (r, :) = knaprs (r,:) + sum(knapr')
knaprsum = knaprsum + knaprs (r, :)

% Storing residual capacities for all models + knapsack capacities
knapr = knapr';
knapr = horzcat(knapr, knapc1(:,2));
```

```
knaprt = vertcat(knaprt , knapr)


r = r - 1;

end

knaprx(1,1) = min (knaprsum);
knaprx(1,2) = mean (knaprsum);
knaprx(1,3) = max (knaprsum);

knaprx;
capsum = sum(sum(cap));
all(cond)

wk1write('knapsac.xls',knaprs)
% wk1write('knapA.xls' , A)
% wk1write('knapC.xls' , c)
wk1write('knapCap.xls' , cap)
wk1write('knapsum.xls' , knaprsum)
wk1write('knaprt.xls' , knaprt)
wk1write('Capsum.xls' , capsum)
```

Appendix B

MATLAB Code for The Tabu-Seach Implemataion in MKAR

```
clear

% # of items and knapsacks
n = input ('          enter total number of items: ');
disp('  ')
m = input ('            enter number of knapsacks: ');
disp('  ')
r = 1  %input ('     enter number of replications/run: ');
disp('  ')
knaprt = [];

knaprs = zeros (r,10); % storage for knapsack residual space
cap = zeros (r,m); % storage for knapsack capacity
cond = zeros (1,r); % storage for conditions
knaprsum = zeros (1,10); % storage for sum of knapsack unutilized
space


while r > 0

% generating item weights
a = ceil(random('unif', 10, 100, n, 1));
w = a;
w = sort (w);  % item weights sorted in ascending order

% generating assignment restriction
b = ceil(random('unif', 0, m, n, m));

% f = item weights
% f = -1 * w;

% generating profits
% a = ceil(random('unif', 10, 100, n, 1));
% p = a;

a1 = 0.4 * (sum(w)/m)
a2 = 0.6 * (sum(w)/m)

% generating similar knapsack capacities
a = (random('unif', a1, a2, m-1, 1))
a (m, 1) = 0.5 * sum(w) - sum (a)
c = ceil(a)
c= sort (c)  % knapsack capacities sorted in ascending order

cap (r,:)=c'

if max (w) <= min (c) & min (c) >= min (w) & sum (w) > max (c);
   cond (1,r) = 1
end

% b = capacities
% b = c;

% generating A
```

```
A = [];
for i = 1 : m
    A = [A ; w'];
end

% storing the items sorted in ascending order
item1 = zeros(n,2);
for i = 1 : n
    item1 ( i, 1) = i;
    item1 ( i, 2) = w ( i, 1);
end
item1; % items sorted in ascending order

% adding knapsack restrictions to items sorted in ascending order
d1 = [item1 b];

w = w(n:-1:1); % item weights sorted in descending order

% storing the items sorted in descending order
item2 = zeros(n,2);
for i = 1 : n
    item2 ( i, 1) = i;
    item2 ( i, 2) = w ( i, 1);
end
item2; % items sorted in descending order

% adding knapsack restrictions to items sorted in descending order
d2 = [item2 b];

% storing the knapsacks sorted in ascending order
knapc1 = zeros(m,2);
for i = 1 : m
    knapc1 ( i, 1) = i;
    knapc1 ( i, 2) = c ( i, 1);
end
knapc1; % knapsacks sorted in ascending order

c = c(m:-1:1);  % knapsack capacities sorted in descending order
% storing the knapsacks sorted in descending order
knapc2 = zeros(m,2);
for i = 1 : m
    knapc2 ( i, 1) = i;
    knapc2 ( i, 2) = c ( i, 1);
end
knapc2; % knapsacks sorted in descending order

% residual capacities storage
knapr = zeros (10,m);


% MODEL 1

% storage location for knapsack assignment
aknap1 = [zeros(n ,1) item1 zeros(n ,1) b];
knap1 = knapc1;
```

```
% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if (aknap1 ( i, 3) <= knap1 ( j, 2)) & (any(knap1(j,1) ==
aknap1(i,5:4+m)))
            aknap1 ( i, 1) = knap1 ( j, 1);
            knap1 ( j, 2) = knap1 ( j, 2) - aknap1 ( i, 3);
            aknap1 ( i, 4) = knap1 ( j, 2);
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end

aknap1;

for i = 1 : m
    knapr (1, i) = knap1 (i, 2);
end
aknap1 % model 1 knapsack assignment

% MODEL 2

% storage location for knapsack assignment
aknap2 = [zeros(n ,1) item1 zeros(n ,1) b];
knap2 = knapc1;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if (aknap2 ( i, 3) <= knap2 ( j, 2)) & (any(knap2(j,1) ==
aknap2(i,5:4+m)))
            aknap2 ( i, 1) = knap2 ( j, 1);
            knap2 ( j, 2) = knap2 ( j, 2) - aknap2 ( i, 3);
            aknap2 ( i, 4) = knap2 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end
```

```
aknap2;

for i = 1 : m
    knapr (2, i) = knap2 (i, 2);
end
aknap2; % model 2 knapsack assignment


% MODEL 3

% storage location for knapsack assignment
aknap3 = [zeros(n ,1) item1 zeros(n ,1) b];
knap3 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if (aknap3 ( i, 3) <= knap3 ( j, 2)) & (any(knap3(j,1) ==
aknap3(i,5:4+m)))
            aknap3 ( i, 1) = knap3 ( j, 1);
            knap3 ( j, 2) = knap3 ( j, 2) - aknap3 ( i, 3);
            aknap3 ( i, 4) = knap3 ( j, 2);
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap3;

for i = 1 : m
    knapr (3, i) = knap3 ((m+1)-i, 2);
end
aknap3; % model 3 knapsack assignment


% MODEL 4

% storage location for knapsack assignment
aknap4 = [zeros(n ,1) item1 zeros(n ,1) b];
knap4 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
```

```
        if (aknap4 ( i, 3) <= knap4 ( j, 2)) & (any(knap4(j,1) ==
aknap4(i,5:4+m)))
            aknap4 ( i, 1) = knap4 ( j, 1);
            knap4 ( j, 2) = knap4 ( j, 2) - aknap4 ( i, 3);
            aknap4 ( i, 4) = knap4 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap4;

for i = 1 : m
    knapr (4, i) = knap4 ((m+1)-i, 2);
end
aknap4; % model 4 knapsack assignment


% MODEL 5

% storage location for knapsack assignment
aknap5 = [zeros(n ,1) item2 zeros(n ,1) b];
knap5 = knapc1;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if (aknap5 ( i, 3) <= knap5 ( j, 2)) & (any(knap5(j,1) ==
aknap5(i,5:4+m)))
            aknap5 ( i, 1) = knap5 ( j, 1);
            knap5 ( j, 2) = knap5 ( j, 2) - aknap5 ( i, 3);
            aknap5 ( i, 4) = knap5 ( j, 2);
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end

aknap5;

for i = 1 : m
```

```
    knapr (5, i) = knap5 (i, 2);
end
aknap5; % model 5 knapsack assignment


% MODEL 6

% storage location for knapsack assignment
aknap6 = [zeros(n ,1) item2 zeros(n ,1) b];
knap6 = knapc1;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if (aknap6 ( i, 3) <= knap6 ( j, 2)) & (any(knap6(j,1) ==
aknap6(i,5:4+m)))
            aknap6 ( i, 1) = knap6 ( j, 1);
            knap6 ( j, 2) = knap6 ( j, 2) - aknap6 ( i, 3);
            aknap6 ( i, 4) = knap6 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap6;

for i = 1 : m
    knapr (6, i) = knap6 (i, 2);
end
aknap6; % model 6 knapsack assignment


% MODEL 7

% storage location for knapsack assignment
aknap7 = [zeros(n ,1) item2 zeros(n ,1) b];
knap7 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if (aknap7 ( i, 3) <= knap7 ( j, 2)) & (any(knap7(j,1) ==
aknap7(i,5:4+m)))
            aknap7 ( i, 1) = knap7 ( j, 1);
            knap7 ( j, 2) = knap7 ( j, 2) - aknap7 ( i, 3);
            aknap7 ( i, 4) = knap7 ( j, 2);
```

```
            if j < m
                k = j + 1;
            else
                k = 1;
            end
            break
        elseif j < m
            continue
        end
    end
end

aknap7;

for i = 1 : m
    knapr (7, i) = knap7 ((m+1)-i, 2);
end
aknap7; % model 7 knapsack assignment


% MODEL 8

% storage location for knapsack assignment
aknap8 = [zeros(n ,1) item2 zeros(n ,1) b];
knap8 = knapc2;

% assignment of items to knapsacks
k = 1;
for i = 1: n
    for j = k:m
        if (aknap8 ( i, 3) <= knap8 ( j, 2)) & (any(knap8(j,1) ==
aknap8(i,5:4+m)))
            aknap8 ( i, 1) = knap8 ( j, 1);
            knap8 ( j, 2) = knap8 ( j, 2) - aknap8 ( i, 3);
            aknap8 ( i, 4) = knap8 ( j, 2);
            if j < m
                k = j;
            else
                k = 1;
            end
            break
        elseif j == m
            k = 1;
        end
    end
end

aknap8;

for i = 1 : m
    knapr (8, i) = knap8 ((m+1)-i, 2);
end
aknap8; % model 8 knapsack assignment
```

```
% MODEL 9

% storage location for knapsack assignment
aknap9 = [zeros(n ,1) item1 zeros(n ,1) b];
knap9 = knapc2;

% assiginment of items to knapsacks
for i = 1: n
    knap9 = sortrows(knap9,[2]);
    for j = m:-1:1
        if (aknap9 ( i, 3) <= knap9 ( j, 2)) & (any(knap9(j,1) ==
aknap9(i,5:4+m)))
            aknap9 ( i, 1) = knap9 ( j, 1);
            knap9 ( j, 2) = knap9 ( j, 2) - aknap9 ( i, 3);
            aknap9 ( i, 4) = knap9 ( j, 2);
            break
        elseif j > 1
            continue
        end
    end
end

aknap9;

for i = 1 : m
    knapr (9, i) = knap9 ((m+1)-i, 2);
end
aknap9; % model 9 knapsack assignment


% MODEL 10

% storage location for knapsack assignment
aknap10 = [zeros(n ,1) item2 zeros(n ,1) b];
knap10 = knapc2;

% assignment of items to knapsacks
for i = 1: n
    knap10 = sortrows(knap10,[2]);
    for j = m:-1:1
        if (aknap10 ( i, 3) <= knap10 ( j, 2)) & (any(knap10(j,1) ==
aknap10(i,5:4+m)))
            aknap10 ( i, 1) = knap10 ( j, 1);
            knap10 ( j, 2) = knap10 ( j, 2) - aknap10 ( i, 3);
            aknap10 ( i, 4) = knap10 ( j, 2);
            break
        elseif j > 1
            continue
        end
    end
end

aknap10;

for i = 1 : m
```

```
    knapr (10, i) = knap10 ((m+1)-i, 2);
end
aknap10; % model 10 knapsack assignment


% Knapsack residual capacities for all models
knapr

knaprs (r, :) = knaprs (r,:) + sum(knapr')
knaprsum = knaprsum + knaprs (r, :)

% Storing residual capacities for all models + knapsack capacities
knapr = knapr';
knapr = horzcat(knapr, knapc1(:,2));
knaprt = vertcat(knaprt , knapr)

knaprsu = (sum(knapc1(:,2))- knaprs)/(sum(knapc1(:,2)))*100

r = r - 1;

end

knaprx(1,1) = min (knaprsum);
knaprx(1,2) = mean (knaprsum);
knaprx(1,3) = max (knaprsum);

knaprx;
capsum = sum(sum(cap));
all(cond)

%wk1write('knapsac.xls',knaprs)
% wk1write('knapA.xls' , A)
% wk1write('knapC.xls' , c)
%wk1write('knapCap.xls' , cap)
%wk1write('knapsum.xls' , knaprsum)
%wk1write('knaprt.xls' , knaprt)
%wk1write('Capsum.xls' , capsum)


r1 = 1;
r2 = 10;
jnaprs = zeros (r1,10); % storage for knapsack residual space
jnaprsum = zeros (1,10); % storage for sum of knapsack unutilized
space
jknaprs = knaprs;


while r2 > 0

[r3,r4] = min ( jknaprs );  % maximum utilized capacity model
jknaprs ( 1, r4 ) = inf;
```

```
if r4 == 1 % is it model 1?
% MODEL 1

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap1;
ajnap = aknap1;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
```

```
    jnapr (1, i) = jnap1 (i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 2 % is it model 2?
% MODEL 2

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap2;
ajnap = aknap2;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
```

```
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (2, i) = jnap1 (i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 3 % is it model 3?
% MODEL 3

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap3;
ajnap = aknap3;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end
```

```
        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (3, i) = jnap1 ((m+1)-i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 4 % is it model 4?
% MODEL 4

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap4;
ajnap = aknap4;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
```

```
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (4, i) = jnap1 ((m+1)-i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 5 % is it model 5?
% MODEL 5

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap5;
ajnap = aknap5;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;
```

```
% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (5, i) = jnap1 (i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 6 % is it model 6?
% MODEL 6
```

```
jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap6;
ajnap = aknap6;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (6, i) = jnap1 (i, 2);
end
```

```
jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 7 % is it model 7?
% MODEL 7

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap7;
ajnap = aknap7;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
```

```
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (7, i) = jnap1 ((m+1)-i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 8 % is it model 8?
% MODEL 8

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap8;
ajnap = aknap8;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
```

```
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (8, i) = jnap1 ((m+1)-i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 9 % is it model 9?
% MODEL 9

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap9;
ajnap = aknap9;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation

while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
```

```
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (9, i) = jnap1 ((m+1)-i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end


if r4 == 10 % is it model 10?
% MODEL 10

jnapc1 = knapc1;
jnapc2 = knapc2;
jnap = knap10;
ajnap = aknap10;
jnap1 = sortrows(jnap,[2]);
ajnap1 = ajnap;
jnap2 = jnap;
ajnap2 = ajnap;
m1 = m;


% tabu search implementation
```

```
while m1 > 0
    if jnap1(m1,2)==0
        m1 = m1 - 1;
        continue
    else
        for i = 1 : n
            if ajnap1(i,1) == jnap1 (m1,1)
                ajnap1(i,1) = 0;
                ajnap1(i,4) = 0;
            end
        end

        for i = 1:m
            if jnap1(m1,1)==jnapc2(i,1)
                jnap1(m1,2)=jnapc2(i,2);
            end
        end

        for i = 1:n
            if (ajnap1 ( i, 3) <= jnap1 ( m1, 2)) & (ajnap1(i,1)==0)
                ajnap1 ( i, 1) = jnap1 ( m1, 1);
                jnap1 ( m1, 2) = jnap1 ( m1, 2) - ajnap1 ( i, 3);
                ajnap1 ( i, 4) = jnap1 ( m1, 2);
            end
        end

        if jnap1(m1,2)< jnap(m1,2)
            ajnap = ajnap1;
            jnap(m1,2) = jnap1(m1,2);
        else
            ajnap1 = ajnap;
            jnap1(m1,2) = jnap(m1,2);
        end

    end
    m1 = m1 - 1;
end

for i = 1 : m
    jnapr (10, i) = jnap1 ((m+1)-i, 2);
end

jnap1;
jnap2 = [jnap2 jnap1(:,2)];
ajnap2 = [item2 ajnap2(:,1) ajnap1(:,1)];

end

r2 = r2 - 1;

end

% Knapsack residual capacities for all models
jnapr;
```

```
jnaprs (r1, :) = jnaprs (r1,:) + sum(jnapr');
jnaprsum = jnaprsum + jnaprs (r1, :);

jnaprsu = (sum(knapc1(:,2))- jnaprs)/(sum(knapc1(:,2)))*100;
% rsu = [knaprsu; jnaprsu];
% for i = 1:10
%     rsu (3,i) = 100*((rsu(2,i)/rsu(1,i))/rsu(1,i));
% end
% rsu = rsu'

rsu1 = [];
for i = 1 : 5
    rsu1 = [rsu1 knaprsu(1,i) jnaprsu(1,i)];
end
rsu2 = [];
for i = 6 : 10
    rsu2 = [rsu2 knaprsu(1,i) jnaprsu(1,i)];
end

rsu = [rsu1; rsu2]

wk1write('p1.xls',rsu);
```

Appendix C

MATLAB Code for the SKPPC

```
clear

% # of items and knapsacks
n = input ('          enter total number of packages: ');
disp('  ')
m = input ('            enter number of inspectors: ');
disp('  ')
r = input ('     enter number of replications/run: ');
knaprt = [];
knaprave = size(r , 2);


while r > 0

% generating item weights
a = ceil(random('unif', 10, 100, n, 2));
w1 = a;
w2 = ceil(mean(sum(a)));
% w = sort (w);  % item weights sorted in ascending order

% generating item outcomes
p = random('unif', 0, 1, n, 1);


for i = 1 : n
    if p(i,1) < 0.8
        b(i) = 2; % select the package with 0.8 probability
    else
        b(i) = 1; % select the package with 0.2 probability
    end
end

a1 = 0.4 * (w2/m);
a2 = 0.6 * (w2/m);

% generating similar knapsack capacities
a = (random('unif', a1, a2, m-1, 1));
a (m, 1) = 0.5 * w2 - sum (a);
c = ceil(sum(a)/m);
% c= sort (c);  % equal knapsack capacities

% selecting items to be assigned
for i = 1 : n
    if b(i) == 1;
        w (i) = w1 (i , 1);
    else
        w (i) = w1 (i , 2);
    end
end

% generating A
A = [];
for i = 1 : m
    A = [A ; w];
end
```

```
w = w';

% storing the items
% item1 = zeros(n,2);
for i = 1 : n
    item1 ( i, 1) = i;
    item1 ( i, 2) = w (i);
end


% storing the knapsack capacities
knapc1 = zeros(m,2);
for i = 1 : m
    knapc1 ( i, 1) = i;
    knapc1 ( i, 2) = c ;
end


% storage location for knapsack assignment
aknap = [zeros(n ,4) b'];
knap = knapc1;

% assignment of items to knapsacks
for i = 1: n
    knap = sortrows(knap,[2]);
    for j = m:-1:1
        if (item1 ( i, 2) <= knap ( j, 2))
            aknap ( i, 1) = knap ( j, 1);
            aknap ( i, 2) = item1 ( i, 1);
            aknap ( i, 3) = item1 ( i, 2);
            knap ( j, 2) = knap ( j, 2) - item1 ( i, 2);
            aknap ( i, 4) = knap ( j, 2);
            break
        elseif j > 1
            continue
        else
            aknap ( i, 2) = item1 ( i, 1);
            aknap ( i, 3) = item1 ( i, 2);
        end
    end
end

knaprt = ones(m,2);

for i = 1 : m
    for j = 1 : n
        if aknap(j,1) == i & aknap(j,5) == 1
            knaprt(i,1) = knaprt(i,1)*0.2;
            knaprt(i,2) = knaprt(i,2) + aknap(j,3);
        elseif aknap(j,1) == i & aknap(j,5) == 2
            knaprt(i,1) = knaprt(i,1)*0.8;
            knaprt(i,2) = knaprt(i,2) + aknap(j,3);
        end
    end
```

```
    knaprt(i,2) = knaprt(i,2) - 1;
    knapr(1,i) = knaprt(i,1) * ( c - knaprt(i,2));
end

knaprs = sum(knapr);

up = sum(knaprt(:,2))/sum(knapc1(:,2));

knaprave(r,1) = knaprs;
knaprave(r,2) = up;

r = r - 1;

end

knaprave;
s = mean(knaprave);

s1 = s(1,1);
s2 = 100*s(1,2);


['The expected penalty cost is ' num2str(s1)]

['The percentage of utilization is ' num2str(s2)]
```

Appendix D

MATLAB Code for the Inspection Problem

```
clear all;

% # of items and knapsacks capacities
n1 = 15; %input ('          enter number of packagess to be assigned:
');
disp('  ')
m = 3; %input ('          enter number of inspectors: ');
disp('  ')
k = 18; %input ('            enter amount of resources available to
inpectors: ');
disp('  ')
w1 = 16; %input ('            enter processing time of item type 1:
');
disp('  ')
w2 = 1; %input ('            enter processing time of item type 2: ');
disp('  ')
p1 = 0.2; %input ('            enter probability of item type 1: ');
disp('  ')
p2 = 1 - p1; %input ('            enter probability of item type 2:
');
disp('  ')
d1 = 0.5; %input ('            enter under-utilization penalty: ');
disp('  ')
d2 = 1 - d1; %input ('            enter over-utilization penalty: ');

N1 = m;
B1 = [];
E = [];
F = [];
G = [];

for n = 3 : n1

    N2 = n;
    V1 = (0:n);
    V2 = [w1 w2];
    V3 = [p1 p2];

    A1 = combn(V1, N1);
    A2 = combn(V2, N2);
    A3 = combn(V3, N2);

    clear V*;

    [r,c] = size(A1);

    X = A1;
    B = [];

    for i = 1 : r
        if sum(X(i,:)) == n
            B = [B ; X(i,:)];
        end
    end
```

```
    B = sort(B,2);
    B = unique(B,'rows');
    B1 = [B1 ; B];

    [r1 c1] = size(B);

    for i = 1 : r1
        for j = 1 : c1
            if B(i,j) == 0
                E1(i,j) = sum(k*prod(A3,2));
            else
                Y = A2;
                Y(:,1:n-B(i,j))=[];
                E1(i,j) = sum(abs(k-sum(Y,2)).*prod(A3,2));
            end
        end
    end

    E2 = 0.5*sum(E1,2);
    E3 = min(E2);
    E = [E ; E2];
    F = [F ; E3];
    G = [G; n];

end

E;
F;
a = min(E);
b = find(E == a);
clc;
disp ('the minimum penalty cost is'), disp(a)
disp ('from the assignment'), disp(B1(b,:))
BE = [B1 E];
plot(G,F);
xlabel('# of items inspected');
ylabel('penalty cost');
wk1write('AC.xls',F);
```