T-4107

A HEURISTIC ALGORITHM FOR SOLVING ECONOMIC MODELS FORMULATED AS 0-1 INTEGER PROGRAMMING PROBLEMS

by

Ahmed A. Zugail

A thesis submitted to the Faculty and Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Mineral Economics).

Golden, Colorado

Date 9/20/91

Signed: ______________________
Ahmed A. Zugail

Approved: ______________________
Dr. R.E.D. Woolsey
Thesis Advisor

Golden, Colorado

Date 9/20/91

____
Dr. Roderick G. Eggert
Associate Professor and
Acting Head
Mineral Economics Department

ABSTRACT

In this work, a new heuristic algorithm for a class of 0-1 integer programming problems is presented. The algorithm is designed to solve a class of economic models formulated as:

$$\text{Max } Z = \sum_{j=1}^{n} c_j x_j \qquad j = 1, 2, \ldots, n$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i = 1, 2, \ldots, m$$

$$c_j, b_i \geq 0$$

$$x_j = 0 \text{ or } 1$$

The algorithm can be used for other applications if formulated as above; i.e., minimization models that can be converted to this form. It is composed of two parts: the ranking algorithm and the substitution algorithm. In the first part, using a new approach, the variables are ordered and the initial solution is obtained. This ranking order is maintained throughout the procedure of the algorithm. The initial solution represents the largest feasible combination of variables allowed by the problem, and the second part of the algorithm tries to improve on this solution by searching for smaller combinations

with higher objective value. The maintenance of the ranking order throughout the procedure of the algorithm is essential in facilitating and accelerating the search process.

The algorithm is very effective and simple and can appeal to decision makers, practitioners, academicians, and students of operations research and management science. The algorithm was tested using 36 problems representing a wide variety of economic applications ranging in size from 6 to 105 variables on an IBM-compatible PC. The results of computational experience were compared to that of exact algorithms, heuristic algorithms, and a commercial package. In almost every case, our heuristic algorithm outperformed them from accuracy and running-time perspectives. The algorithm averaged more than 99.80% of optimality with its most accurate versions, and more than 98.10% with its least accurate versions. To the knowledge of the author, the algorithm also solved large problems that have never been solved on a PC.

There is still room for improvement in the algorithm in order for it to be more effective. Suggestions for further areas of research are provided.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

IN THE NAME OF ALLAH THE MOST GRACIOUS

THE MOST MERCIFUL

ACKNOWLEDGEMENTS

All praise is due Allah the most beneficent, the most compassionate, and peace be upon his prophets.

I wish to express my sincere gratitude and appreciation to King Fahad University of Petroleum and Minerals in the Kingdom of Saudi Arabia for the full scholarship and generous financial support throughout my graduate study in the United States of America.

I wish also to express my special appreciation and thanks to Dr. R. E. D. Woolsey, my dissertation advisor, for his guidance and encouragement. I have profited from his wisdom and experience.

I would like to thank the members of my dissertation committee, Dr. Ruth Maurer, Dr. Donald Dickinson, Dr. Robert Drury, and Dr. Robert Patrick for their comments and valuable suggestions.

I wish also to thank my friend, Rashid Al-Hamoodah, who wrote an efficient computer code of the algorithm presented in this work. My gratitude is due to Owen Dull, Joe Huber, Lita Dunham, and Betty Mahaffey.

Finally, I cannot sufficiently express my love, appreciation, and gratitude to my parents and my wife for their love, sacrifices, help, and encouragement, not only throughout my study but also throughout my life.

TO MY PARENTS

Chapter 1
INTRODUCTION

Optimization under the requirement of integer answers is as old as humans on this earth. Man had to allocate his limited resources efficiently between his needs of food, clothes, shelter, and leisure. He had to know how many camels, horses, swords, arrows, rooms, slaves, etc. he needed to satisfy these needs. However, it was not until 35 years ago that integer programming (IP) was established as a separate field of mathematical programming receiving special attention of extensive theoretical and practical research. One reason for this attention was the tremendous number of applications that can be formulated as integer programming models, which includes economic, social, engineering, scientific, military, and even political applications.

Zionts (1974) stated that any deterministic problem which can be precisely described in quantitative terms can be formulated as accurately as desired as a mixed integer programming problem. This attention coincided with the rise of the computer technology which made small integer programming problems accessible to solution in a matter of minutes or seconds whereas hours of hand calculations had been spent to solve them.

Another reason was the fascinating challenges this class of problems presented to mathematicians, researchers, and practioners of operations research and management science. These challenges rose from establishing the theoretical background and developing techniques to solve the wide variety of models under this class of problems efficiently. A fourth reason is the fact that many important special conditions, which characterize most resource allocation problems, can not be dealt with practically without the use of integer variables (Plane and McMillan, 1971).

The use of linear programming (LP) methods to solve integer programming problems and then rounding the results to the nearest integer has only been feasible in problems where decision variables need to be large numbers such as in the production of automobiles. In other cases, rounding the LP solution can either produce a solution far from optimal or an infeasible one. For this reason, special techniques to solve IP problems that produce discrete value answers had to be developed.

The higher degree of difficulty of integer programming problems compared to that of LP rises from the nonlinear nature of all or some of the decision variables involved. Since these variables are discrete, it is required that IP techniques need to deal with the integer space directly, as in search methods, or indirectly by always redefining the corner points of the feasible integer space, as in the cutting methods.

At this time there is no "universal" method that solves all integer programming problems effectively and efficiently comparable to the simplex method of linear programming. Present methods either perform inconsistently or take impractical, or unaffordable, time to reach an optimal or near optimal solution. This is mainly due to the following facts.

The first fact is that there is a wide range of models and different structures of integer programming problems. While some methods work very well with some IP models, they behave poorly with others. The second is that the practicality of a method requires that it can solve or approximate the solution of large real-world problems in an economically feasible time. The fact of the matter is that real-world applications are usually large problems. This would translate into huge numbers of points to be examined in order to solve these problems. For example, a 20 0-1 variable problem means that there are $2^{20} > 1,000,000$ possible combinations to be considered. It can be quickly seen that problems with 50, 100, 1,000, and more variables soon exceed even the fastest computers to attain a solution in a reasonable time.

This led to the development of fast approximate methods, specialized methods that exploit special structures of some models, composite methods, and reduction and preprocessing procedures. Using these tools in combination with exact IP algorithms, LP methods, and experience (according to Woolsey, 1972) aided by the improvement of the digital computer capabilities, prohibitively large problems are now amenable to solution.

The purpose of this dissertation is to present a new, efficient heuristic algorithm for solving a class of economic models formulated as 0-1 IP problems. It is designed to solve problems of the form:

$$\text{Max } Z = \sum_{j=1}^{n} c_j x_j \qquad j = 1, 2, \ldots, n$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i = 1, 2, \ldots, m$$

$$x_j = 0 \text{ or } 1$$

and without any loss of generality it is assumed that c_j, $b_i \geq 0$. Investment analysis, R&D project selection, capital budgeting, and resource allocation problems are examples of this model formulation (Lorie and Savage, 1955 and

Weingartner, 1963). The algorithm can be used alone or in combination with other known IP algorithms. Published real-world R&D project selection problems (Petersen, 1967) are used to test the computational efficiency of the algorithm; the exercise is then expanded to a wider range of applications. It is hoped that the design and the mechanics of this algorithm should be very appealing to managers and decision makers.

A general background about integer programming, its applications, present methods, and computational experiences and difficulties are presented in the second chapter. In the third chapter, the algorithm is described and its mechanics and advantages analyzed in detail. In the fourth chapter, computational experience with the algorithm and comparisons to other algorithms and a commercial package are presented. Finally, comments, extensions, and improvements on the algorithm are suggested in the fifth chapter.

Chapter 2

BACKGROUND

The goal of integer programming is to find integer values for some or all the decision variables involved in order to optimize (minimize or maximize) an objective function subject to a number of constraints. During the 1950's and the 1960's IP was the subject of extensive research because of its fascinating challenges and wide range of applications. Many real-world applications can be formulated as integer programming models. Number of projects to be chosen, warehouses to be built, operators to be scheduled, units to be produced, transportation routes and shipping outlets to be assigned, and so on should all be integers. One may be able to round off or round up linear programming answers for problems that require large numbers, such as the production of computer chips. However, for problems requiring small numbers, this practice has been shown to be a disastrous road and other techniques must be utilized (Woolsey, 1988).

2.1. APPLICATIONS OF INTEGER PROGRAMMING

Integer programming can be classified into three broad categories. All-integer or pure integer, mixed-integer where some of the variables are continuous, and zero-one integer programming where decision variables are limited to the values zero and one. A huge number of real-world applications can be formulated using different models under these classes. However, in general, any integer or mixed-integer programming problem can be written as:

$$\begin{aligned} \text{Max} \quad & Z = CX + HY \\ \text{s.t.} \quad & AX + GY \leq B \\ & X, Y \geq 0 \\ & X \text{ integer} \end{aligned}$$

where C is an n-component row vector, X and Y are n-component column vectors, b is an m-component column vector, A is an m x n matrix, H is an q-component row vector, and G is an m x q matrix. However, this general formulation varies with the different applications of IP. A number of IP models are presented below with their most common applications.

Capital budgeting is concerned with the selection of the most profitable combination of investment projects subject to the available resources. There is a variety of formulations for the capital budgeting problem. These variations depend on the choice of incorporating the concept of risk and expected net present value in the parameters of the objective function (Weingartner, 1963 and 1966). This problem can be represented by the following model:

$$\text{Max } Z = \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i = 1,2, \ldots, m$$

$$x_j = 0 \text{ or } 1 \qquad j = 1, 2, \ldots, n$$

where c_j represents the payoff of project j, a_{ij} is the cost of project j in using resource i, b_i is the available from resource i or the available budget at period i, and $x_j = 0$ or 1 denotes the acceptance or the rejection of project j.

The assignment model represents the problem of assigning people to tasks or limited resources to different jobs in order to maximize profit or efficiency, or to minimize the cost or time needed to complete jobs. Some formulations of this model use 0-1 decision variables. The following model is very common:

$$\text{Min } Z = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1,2, \ldots, m$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad j = 1, 2, \ldots, n$$

$$x_{ij} = 0 \text{ or } 1$$

where $x_{ij} = 1$ if person j is assigned to job i and $x_{ij} = 0$ otherwise, and c_{ij} represents the cost associated with person j doing job i. The first set of constraints make sure that exactly one person does job i, and the second set guarantees that each person does no more than one job. This particular problem when run on an LP code always produces integer solutions due to the structure of the matrix (Woolsey, 1991). Scheduling, matching, and assembly line balancing problems can also be formulated using this model with slight variations (for more details see Plane and Mcmillan, 1971; Taha, 1975; and Nemhauser and Wolsey, 1988).

The knapsack problem is a classical model that received considerable attention in the literature for its wide range of application and for the possibility of converting general IP problems into knapsack-type models. The name "knapsack" is taken from the situation that a hiker needs to maximize the value of the items he takes under limitation of specified weight. Applications of this model range from cargo-loading to project selection and cutting stock problems. The classical single knapsack problem can be written as:

$$\text{Max } Z = \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_j x_j \le b$$

$$x_j = 0 \text{ or } 1$$

In the case of more general integer programming problems, such as the cutting stock problem, this model looks like the following:

$$\text{Min } Z = \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j \ge b_i \qquad i = 1, 2, \ldots, m$$

$$x_j \ge 0 \text{ and integer}, \qquad j = 1, 2, \ldots, n$$

where x_j is the number of standard pieces of a given material to be cut according to pattern j, a_{ij} is the number of pieces cut in a certain length in cutting pattern j, b_i is the number of ordered pieces cut in a certain length, and c_j is the cost associated with each cutting pattern j. In this model the width is assumed to be the same. It is obvious that this problem can be very difficult computationally when there is a large number of variables since x_j is restricted to integer values. This problem was the inspiration for pioneering work in integer programming by Gilmore and Gomory (1961). For more details on the knapsack problem see Weingartner and Ness (1967), Cabot (1970), and Taha (1975).

Set covering, set packing, and set partitioning problems have a wide range of applications. Delivering orders from a warehouse to different destinations each of which receives its order in one delivery, airline crew scheduling, shift scheduling, and facility location are all examples that can be formulated as set covering problems. The variations in the objective function can be to minimize the cost of total delivery or to minimize the number of subjects to be scheduled. The delivery problem can be formulated as follows:

$$\text{Min } Z = \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j \geq a_{i0} \qquad i = 1, 2, \ldots, m$$

$$x_j = 0 \text{ or } 1 \qquad j = 1, 2, \ldots, n$$

$$a_{i0} = 1, \; a_{ij}(j \neq 0) = 0 \text{ or } 1$$

where c_j is the cost of delivering the ith order from warehouse j, and $x_j = 1$ if warehouse j is to be used, and $x_j = 0$ otherwise. The inequalities in the constraint set can be replaced by equalities to ensure that an order is only delivered from one warehouse. This problem has received a lot of attention and efficient algorithms have been developed that exploit its special structure (see Balinski, 1965; Taha, 1975; and Nemhauser and Wolsey, 1988).

One of the classical optimization problems is the traveling salesman problem. The idea is to find the least-cost or the shortest route which will visit each city or destination only once. This problem can be formulated as follows:

$$\text{Min } Z = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij}x_{ij} \qquad \text{where } d_{ij}=\infty \text{ for } i=j$$

$$\text{s.t.} \quad \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, 2, \ldots, n$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, 2, \ldots, n$$

$$x_{ij} = 0 \text{ or } 1 \text{ and } i \neq j$$

solution is a tour

where d_{ij} is the distance between cities i and j, $x_{ij} = 1$ means that the salesman travels directly from city i to city j, and $x_{ij} = 0$ otherwise. The condition that $d_{ij} = \infty$ for $i = j$ ensures that $x_{ij} = 0$ for $i = j$. Ignoring the requirement that the solution should be a tour, this formulation is the same as the assignment model.

An optimal solution of the assignment model is not necessarily going to be a tour. However, a tour solution is guaranteed if all subtours are eliminated. Define $N= \{1,2, . . . ,n\}$ and let S be the set of indices representing the cities associated with a subtour. Then, S is disconnected from U=N-S, that is, no path exits between any of the cities in S and those in U. Thus, a subtour S is eliminated if and only if

$$\sum_{i \epsilon S} \sum_{j \epsilon S} x_{ij} \geq 1$$

since

$$\sum_{i \epsilon S} \sum_{j \epsilon S} x_{ij} = 0$$

means that S is a subtour.

There are $2^n - 2$ possible subtours and hence constraints. This will create a very large problem for a small or even reasonable number of cities ($n = 10$, $2^n - 2 = 1022$). This problem was the subject of extensive research. (For more details, see Dantzig et al., 1954; Little et al., 1963; Greenberg, 1971; Taha, 1975; and Nemhauser and Wolsey, 1988).

The last model that will be presented is the fixed-charge problem. This model represents a number of important applications and is sometimes very difficult to solve. When activities are undertaken, there is usually a fixed cost to be incurred. The total cost of this activity will be the sum of this fixed cost and a variable cost proportional to the level of the activity. The model can be presented as a mixed-integer or an all-integer programming problem. The following is a capital budgeting example.

$$\text{Max } Z = \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t.} \quad \sum_{j=1}^{n} d_{ij} F_j + \sum_{j=1}^{n} a_{ij}\, x_j \le b_i, \qquad i = 1, 2, \ldots, m$$

$$-r_j F_j + x_j \le 0, \qquad j = 1, 2, \ldots, n$$

$$x_j \ge 0 \text{ and integer}$$

$$F_j = 0 \text{ or } 1$$

Where c_j is the payoff per unit of project j, x_j is an integer value of the size of project j limited by the upper bound r_j,

$F_j = 1$ if project j is undertaken and $F_j = 0$ otherwise, d_{ij} is the fixed charge associated with undertaking project j, and b_i is the available budget. There are many applications that can be formulated as fixed-charge problems, such as the plant location and the delivery problems (for more details, see Balinski, 1965; Greenberg, 1971; and Taha, 1975).

2.2. INTEGER PROGRAMMING METHODS

Since the rounding of the LP solution to solve IP problems proved unsatisfactory and suboptimal in almost every case except when the LP solution is integer, special methods to solve these problems had to be developed. There is a number of classifications to integer programming problems. One way to classify these methods is according to the way they attack the problem. Under this classification most integer programming methods can be divided into two major categories, cutting plane methods and enumeration methods.

2.2.1. Cutting Plane Methods

It is possible that the solution using linear programming could result in an integer solution. In 1949, Dantzig noted that the transportation problem always has integer solutions if demands and supplies are given integer values. However, this is not the case for other types of problems and it is very difficult to represent other problems as transportation, assignment, or network flow problems and to properly interpret them (Balinski, 1965).

The idea of the cutting plane is to successively generate new linear constraints that in conjunction with the original constraints will produce an integer solution. This is done by restructuring the feasible region so that the extreme points will satisfy integer requirements. The basic idea of this approach was introduced by Dantzig et al. (1954) in their work on solving the traveling salesman problem, and then by Markowitz and Manne in 1957. There was no guarantee of finiteness of the method until Gomory developed the first finite cutting plane algorithm for the pure integer programming problem in 1958. In 1960, Gomory extended his work to cover mixed integer programming and eliminated the machine round-off error from the first algorithm developing an all integer programming algorithm (Balinski, 1965; Taha, 1975; Kim, 1983).

These algorithms and similar ones are dual methods; that is, no feasible solution for the original problem is found until an optimal solution is found or the algorithm terminates. This is a major disadvantage if calculations are stopped prematurely because it may require a very large number of iterations to solve a problem. To overcome this problem, a primal cutting plane algorithm for general integer programming problems was proposed by Ben-Israel and Charnes in 1962 that requires the solution of a system of linear inequalities in integers as a substep (Balinski, 1965). However, the first finite primal algorithm was developed by Young in 1965, then simplified versions were developed by Glover and Young in 1968 (Taha, 1975).

Reported computational experience by Trauth and Woolsey (1969) comparing different codes of cutting methods for solving 29 pure integer problems suggested that computation time is code dependent and no code is uniformly better for all problems, some cuts are inferior to others, round-off errors are serious, and simple rearrangement of the constraints can convert a computationally difficult problem to a relatively simple one. The general conclusion was that the solution of integer programs by cutting methods is generally difficult and uncertain, especially for large-size problems (i.e., 100 by 100 and more). Computational experience with these methods was not satisfactory except for a primal cutting plane algorithm for the traveling salesman problem by Padberg and Hong (Nemhauser and Wolsey, 1988). For more details about cutting methods see Balinski (1965), Geoffrion and Marsten (1972), Taha (1975), and Nemhauser and Wolsey (1988).

2.2.2. Enumeration Methods

The second method for solving integer programming problems is that of enumeration. These methods are devised to intelligently search the space of all feasible solutions by considering the small portion of it that holds the global optimum. In other words, enumerating all feasible points without considering all of them by considering promising integer points and discarding nonpromising ones.

These methods include branch-and-bound methods for general (mixed) integer programming and implicit enumeration methods mainly for 0-1 integer programming. Some consider the latter as a special class of branch-and-bound methods. The difference between these two subclasses is that branch-and-bound methods rely on pruning by bounds obtained from relaxation or dual solutions, using linear programming, while implicit enumeration methods rely on pruning based on logical tests and inequalities derived from Boolean implications (Nemhauser and Wolsey, 1988).

Branch-and-bound is the term given by Little et al. (1963) to their method for solving the traveling salesman problem. However, it was Land and Doig (1960) who gave the first branch-and-bound algorithm for general integer programming. The idea of the algorithm is simply to apply the simplex method of LP to the problem. If the optimal solution is integer the problem is solved; otherwise, two new problems are created and the original LP solution is considered as an upper (lower) bound for the integer problem. In one problem some variable that has a noninteger value is chosen and restricted to its next lower integer value and to its next higher value in another problem. The process is repeated on each of the new branched problems. During this process most of the feasible solutions are enumerated implicitly and only a few explicitly. The process is terminated when either an optimum solution is found or every branch terminates with an infeasible solution (Balinski, 1965; Taha 1975).

Nemhauser and Wolsey (1988) referred to various refinements and branching rules that were proposed and tested by Beale and Small, Dakin, and Driebeek in the mid-1960s. Since then, several developments were implemented and various strategies of enumeration were explored which are being used in commercial codes. Favorable computational results of the branch-and-bound methods over cutting methods have been reported by Dakin and others. These methods proved that they can be effective in solving large integer problems provided that they are equipped with heuristics (Taha, 1975). However, one of the main disadvantages with these methods is the large amount of computer memory needed. For more details on these methods see Balinski (1965), Lawler and Wood (1966), Taha (1975), and Nemhauser and Wolsey (1988).

Implicit enumeration methods have been tied to 0-1 integer programming; however, general integer programs can be solved using these methods. In general, if the upper bound on an integer variable x is U then, x can be replaced by $x=\sum_{j=0}^{k} 2^j x_j$ where k is the smallest integer such that $2^{k+1} - 1 \geq U$. In the light of the difficulties in solving general integer programming problems using cutting methods, one can say that as long as k does not grow very large, it could be more practical to convert general integer programs to 0-1 integer problems to be solved by implicit enumeration methods. Enumeration techniques proved more effective with binary variables than with general integer problems.

The term implicit enumeration was first given by Geoffrion (1967); however, it was Balas (1965) who developed the first implicit enumeration algorithm for general 0-1 integer programs. It is an enumeration method which examines branches of a combinatorial "tree", by the use of certain tests, to determine if the branches are useless for further search or are possible sources of improvement. The power of the algorithm relies on its ability to implicitly enumerate all the possible combinations by actually testing a small portion of them. Some of the advantages of the algorithm are its simplicity, freedom of rounding errors, and the small memory requirements. Balas described it as an additive algorithm because it uses only addition and subtraction.

The efficiency of the algorithm is restricted to problems with a limited number of variables, because the computer run time increases exponentially with the number of variables. This exponential increase is because, for a problem with n variables, the possible number of solutions for a problem with binary variables is 2^n. A simple comparison between a problem with 5 variables (32 possible combinations) and a problem with 20 variables (more than one million combinations) will give a clear indication about the problem of computational efficiency with larger problems. However, Balas's work was the basis for further refinement and development.

Glover (1965) developed a dual algorithm for 0-1 integer programming in which he introduced the idea of the "surrogate constraint" and suggested extensions to general integer programs. Freeman (1966) reported favorable results with problems of up to 30 variables using a modified version of the algorithm that was produced by Geoffrion. Geoffrion (1967) superimposed Balas's algorithm on Glover's enumeration scheme which produced a reformulation of Balas's algorithm that produced an improved method of scorekeeping with a high rate of exclusion and low storage requirements. Petersen (1967) reported favorable results for problems of up to 50 variables using the reformulated Balas's algorithm in conjunction with modifications emphasizing ordering criteria for examining free variables and more exclusion tests. The above work proved that the efficiency of an algorithm is a function of the power of the exclusion tests so as to eliminate as many nonpromising solution points as possible.

Closely related to Balas's method is a method developed by Lawler and Bell (1966) that makes use of lexicographic ordering. The method was applied successfully to nonlinear objective functions. Special methods to solve mixed 0-1 integer programming problems were developed by Driebeek using penalties, and by Benders and Lemke and Spielberg using partitioning (Taha, 1975).

Because of its wide range of applications, implicit enumeration in conjunction with different modifications and the help of heuristics has been more effective with specially structured problems. Fleischmann (1967) reported favorable results for combinatorial and traveling salesman problems with up to 159 variables. Pierce (1968) reported on computational experience in solving combinatorial programs, where elements of the constraint matrix are zeros and ones, with problems of up to 1,485 variables.

The classical knapsack problem has attracted a lot of attention because it can represent a large number of applications and can serve as a subproblem in the solution of problems such as the generalized assignment and the capacitated facility location problems (Nauss, 1976). Weingartner and Ness (1967) developed algorithms applied within a dynamic programming framework to the multidimensional 0-1 knapsack problem and reported computational improvements with two problems of 28 and 105 variables and two constraints each. Greenberg and Hegerich (1970) presented an algorithm to solve the single knapsack problem that involves searching the nodes of a tree along a single branch at a time. Problems of up to 5,000 variables were solved in approximately four minutes using the IBM 360/67.

Ingergiola and Korsh (1973) developed a reduction algorithm for 0-1 single knapsack problems and reported results on problems of up to 5,000 variables using IBM 370/155. Fayard and Plateau (1975) and Nauss (1976) developed efficient algorithms to solve the 0-1 knapsack problem. Balas and Zemel (1980) used implicit enumeration in conjunction with LP and heuristics to solve randomly generated large single 0-1 knapsack problems of up to 10,000 variables. Maximum reported run time was three seconds on the UNIVAC 1108. Recently Martello and Toth (1988) developed a new algorithm that is particularly effective for large size 0-1 knapsack problems and reported impressive results on solving problems with up to 10,000 variables. For other work with this special class and other 0-1 models see Nemhauser and Wolsey (1988).

Various branch-and-bound algorithms coupled with linear relaxation designed for the uncapacitated facility location problems were reported by Nemhauser and Wolsey (1988). They also described strong cutting planes and fractional cutting planes algorithms that have been developed for a variety of other hard combinatorial problems such as the matching problem, the assignment problem with side constraints, and others.

2.2.3. Approximation Methods

Approximation methods (heuristics) are designed for two main reasons. The first is to enhance the effectiveness of general and special integer programming algorithms. The second is to find "good" solutions that may or may not be optimal that could be fed to exact algorithms when these algorithms fail to find such solutions in an economical time. It has been argued that some heuristics could be considered as common sense tools and have been around for centuries (Nemhauser and Wolsey, 1988). However, a large number of specially designed heuristics are of the direct search type that employ nontrivial ideas and obtain optimal or close to optimal results in an "affordable" time, compared to exact algorithms.

One of the early heuristics designed for discrete optimization is the greedy algorithm. The idea of the algorithm is that the next element to be chosen is the one that gives the greatest increase (decrease) in the value of the objective function for maximization (minimization) problems. In 1956, Kruskal used the greedy heuristic to solve the maximum-weight spanning tree problem exactly. Others have used greedy heuristics to solve the uncapacitated facility location (UFL) problem (Nemhauser and Wolsey, 1988).

Other heuristics are of local search or interchange type. The idea is that the heuristic start with a given feasible solution, and then improve on the solution by making only limited changes. Kuehn and Hamburger (1963), and Manne (1964) used this type of heuristics to solve UFL. Rieter and Sherman (1965) presented interchange algorithms for the traveling salesman problem and implemented a statistical analysis on the results based on random starting solutions (Nemhauser and Wolsey, 1988).

A more recent idea is the pairing of primal and dual heuristics for simultaneous use on the UFL problems and other applications. For more details on these heuristics and their application, see Nemhauser and Wolsey (1988).

Three heuristics that are designed to solve general integer programs are due to Echoles and Cooper, Hillier, and Kochenberger et al. A major drawback of these methods is the assumption that the solution space must have an interior which prevents the explicit or implicit inclusion of equality constraints in the model. One of the advantages of some of these heuristics, however, is that when they work they produce good solutions quickly (Taha, 1975).

The first heuristic by Echoles and Cooper starts with a feasible integer point, then the variable with the best objective function coefficient is increased as much as it is allowed. Then the next best variable is chosen and its value again increased to the maximum limit allowed by feasibility. The process is repeated until the value of the objective function can not be improved. The second phase of the algorithm attempts to find a better solution by manipulating the values of the variables in the current feasible solution (Taha, 1975).

The second heuristic by Hillier (1969) differs from the first one by starting with an integer solution very close to the noninteger optimum produced by the simplex method. In phase two of the algorithm, a search for a good feasible solution starts on the line segments between the starting point and the noninteger optimum. When a feasible integer solution is reached, phase three attempts to improve upon the solution by changing either one or two variables at a time until no further improvement can be made.

The third heuristic by Kochenberger et al. (1974) is stimulated by a 0-1 heuristic proposed by Senju and Toyoda (1968) that uses effective gradient for choice of examined variables. The heuristic starts with an all-zero solution vector, in contrast to Senju and Toyoda's heuristic that starts with an all-one solution, and increments variables one at a time on the basis of the amount of objective function increase per unit of feasibility used. The method terminates when no variable can be increased. This termination step does not allow for backtracking or manipulating the last solution, which is a major disadvantage especially in tightly constrained problems. Computational experience reported on the set of problems presented in Trauth and Woolsey (1969) showed that the heuristic is fast, but it had an 86% average accuracy. Senju and Toyoda presented two large problems of 60 variables by 30 constraints that they solved using their heuristic to a 99.4% average accuracy, and then were solved optimally by Marsten and Morin (1977).

A useful comparative study by Zanakis (1977) on the performance of three heuristics by Senju and Toyoda, Kochenberger et al., and Hillier on 0-1 problems concluded that the Hillier method was the most accurate but the slowest; the Kochenberger et al. method was the fastest (most accurate) in tightly (loosely) constrained problems; and the Senju and Toyoda method was the fastest and the most accurate in general.

The last heuristic to be presented is the pivot and complement heuristic for 0-1 programming by Balas and Martin (1980). The procedure starts by solving linear programs; then performs a sequence of pivots to put all slacks in the basis at a minimal cost; and finally, it attempts to improve the 0-1 solution by a local search based on complementing certain sets of 0-1 variables. The study reported favorable computational results on the UNIVAC 1108 of 92 tested problems with 5 to 200 constraints and 20 to 900 variables.

Attempts to solve sparse large-scale 0-1 programming problems made by Crowder et al. (1983) confirmed that a combination of problem pre-processing, cutting planes, and clever branch-and-bound techniques can be very effective in solving these problems in a reasonable time. They reported impressive computational results using the IBM 370/168 machine of ten real-world industrial applications with up to 2,750 variables. Another study that is on the same line as the previous one and used schemes to take advantage of special structures present in the tested models is one by Johnson et al. (1985).

These, and other studies suggest that, most of the time, the best approach to solve IP problems, especially real-world applications, is to have an insight into how to attack the problem before using the computer to apply known IP methods or use commercial packages; a point emphasized by Woolsey (1972). The fact that there is no all-purpose method to solve IP problems is the driving force behind the search for effective ways to deal with this class of mathematical programming. Solving integer programming problems is an art.

Chapter 3

THE ALGORITHM

One of the main reasons for my coming up with this algorithm is that I have been attracted to 0-1 integer programming since taking the IP course with Dr. Woolsey in 1988. Trying a lot of small problems (ranging from 5 to 20 variable problems) by hand led me to this work. A second reason is that, except for specialized methods designed for specially structured problems, there is a lack in the recent literature about computational efficiency in general 0-1 programming and the development of new algorithms and heuristics that deal with these problems. Another reason is that decision makers and managers are usually uncomfortable with answers that they cannot read or do not understand (Woolsey, 1972). Our algorithm tries to overcome these problems and to appeal to as many concerned people as possible. The algorithm presented in this work is a simple non-redundant effective enumeration heuristic procedure that solves a variety of 0-1 integer programming economic models under the general form:

$$\text{Max } Z = \sum_{j=1}^{n} c_j x_j \qquad j = 1, 2, \ldots, n$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij}x_j \le b_i \qquad i = 1, 2, \ldots, m$$

$$x_j = 0 \text{ or } 1$$

Without any loss of generality it is assumed that c_j, $b_i \ge 0$, though some problems with $b_i < 0$ were solved optimally. Since most of economic models are to maximize profit or revenue, the work with the maximization form is another reason for developing this algorithm (minimization models have to be converted to maximization form in order to be solved using this algorithm).

The maximization necessity of this model triggered the idea of starting with the largest feasible combination of variables allowed by the problem as an initial solution. This produces two desired possibilities. The first possibility is that the initial solution is optimum and the enumeration process to check this result will terminate quickly. The second possibility is that if this initial solution is not optimum, most of the time it is a good starting solution, depending on the structure of the problem. The searching process to improve upon this solution by looking for smaller combinations with better objective function values can produce impressive results in a very short time.

The algorithm is composed of two parts. The first part is called the ranking algorithm and the second part is the substitution algorithm. In a new procedure, the ranking algorithm produces a good initial solution which is used by the substitution algorithm to find an optimal or very close to optimal solution. The algorithm has produced impressive results in comparison to other algorithms in the literature. The algorithm and a number of its versions are described in detail below.

3.1. THE RANKING ALGORITHM

1) For each constraint, sort x_j from left to right in increasing order of a_{ij} size; in case of ties put first the variable with the smallest $\sum_{i=1}^{m} a_{ij}$; in case of ties put first the variable with the largest c_j; break ties arbitrarily.

2) For each constraint, find the maximum number of variables that a constraint can accommodate by summing a_{ij} according to the sort of the previous step. Call this number HDC (highest degree of combinations). The HDC number for the problem is the smallest of the HDC numbers for the constraints. Sort the constraints from top to bottom in increasing order of their HDC number.

3) For each x_j, sum the sorted positions in all constraints, then sort the variables in increasing order of their position sum, and this will be the final order that will be used and maintained throughout the procedure of the algorithm. Break the final order into two groups by counting variables from the left of the final order equal to the problem's HDC number, the IN group (x_j=1) is formed. The remaining variables constitute the OUT group (x_j=0).

4) Check feasibility of the IN group; if it is feasible then find the value of the objective function (Z) and this is the initial solution to be used in the substitution algorithm; otherwise, drop the rightmost variable from the IN group and add it to the left of the OUT group and then check for feasibility. Repeat the process until feasibility is attained.

3.2. THE SUBSTITUTION ALGORITHM

Eliminate from substitution comparisons by circling any OUT group variable that has an objective function value less than the objective function values of all IN group variables, and any IN group variable that has an objective function value greater than the objective function values of all OUT group variables. Go to Step 1.

1) Starting with one-for-one substitution level (N=1, a variable from the OUT group substitutes for another from the IN group), compare the rightmost free variable (not circled or underlined) from the OUT group (x_k) to the rightmost variable, or variables for higher substitution levels N>1, from the IN group (x_r):

a) If $c_k > \sum_{r=1}^{N} c_r$ (c-test), and if $a_{ik} - \sum_{r=1}^{N} a_{ir} \leq y_i$ (a-test) where $y_i = b_i - \Sigma\, a_{ir}$, substitute x_k for x_r(s), place each in their new groups according to their position sum, find Z, and take the next x_k; otherwise, take the next x_r(s) by moving to the left of the IN group and compare to x_k.

b) If $c_k = \sum_{r=1}^{N} c_r$, and if $a_{ik} < \sum_{r=1}^{N} a_{ir}$ (strict inequality is necessary to prevent cycling), substitute x_k for x_r(s), place each in their new groups according to their position sum, find Z, and take the next x_k; otherwise, take the next x_r(s) and compare to x_k.

c) if $c_k < \sum_{r=1}^{N} c_r$, do not substitute and take the next x_r(s) and compare to x_k.

2) After each substitution, clear any single or double-underlined x_k(s) and x_r(s), circle any free x_k with a c_k < every c_r, and circle any free x_r with a c_r > every c_k, which means that these variables will not be considered in future comparisons. Set N=1, go to step 1.

3) If an x_k fails every c-test (i.e., $c_k<c_r(s)$) in some substitution level, then double-underline it. This means that it should not be considered in higher substitution levels.

4) If an x_k fails every a-test for every c-test passed (an x_k cannot enter the a-test unless it passes the c-test; i.e., $c_k>c_r(s)$ or $c_k=c_r(s)$) in some substitution level, then if the smallest $c_r > (c_k/(N+1))$ or the sum of the N+1 smallest $c_r(s)>c_k$, double-underline x_k and other x_k's with smaller c_k; otherwise, single-underline it, which means that it should be considered in higher substitution levels.

5) If there is no free x_k and any is single-underlined and there are free $x_r(s)$, then go to step 6; otherwise, go to step 7.

6) Single-underline all x_r's that satisfy this condition: if the smallest N c_r's + the largest free c_r > the largest single-underlined c_k, then underline the largest free x_r (which means it should not be considered in higher substitution levels). Then clear single-underlined x_k's and go to the next substitution level (N=N+1) and go to step 1 for N>1.

7) If every x_k is circled or double-underlined, or every x_r is circled or underlined then, clear all variables in

both groups and check if any x_k can be added to the IN group without being infeasible. If not, the last solution is the best which can be obtained, STOP. Otherwise, add the best combination possible and STOP. The last solution is the best which can be obtained.

An example describing the procedure of the algorithm is given in Appendix A. Also, a block diagram of the algorithm is shown in Figure 1.

3.3. DISCUSSION

The presented algorithm is a primal search algorithm in the sense that it starts with a feasible solution and if the search is terminated suddenly, a feasible solution will still be at hand. Normally, present enumeration or search algorithms, depending on the initial solution, have to check larger and smaller combinations of variables in search of the optimal solution. This algorithm is designed to search in one direction only, that is from the largest combination of variables allowed by the problem down towards smaller combinations looking for improvement in the value of the objective function. The largest feasible combination of variables is obtained from the ranking algorithm where the variables are divided into two groups, namely the IN group (x_j=1) and the OUT group (x_j=0).

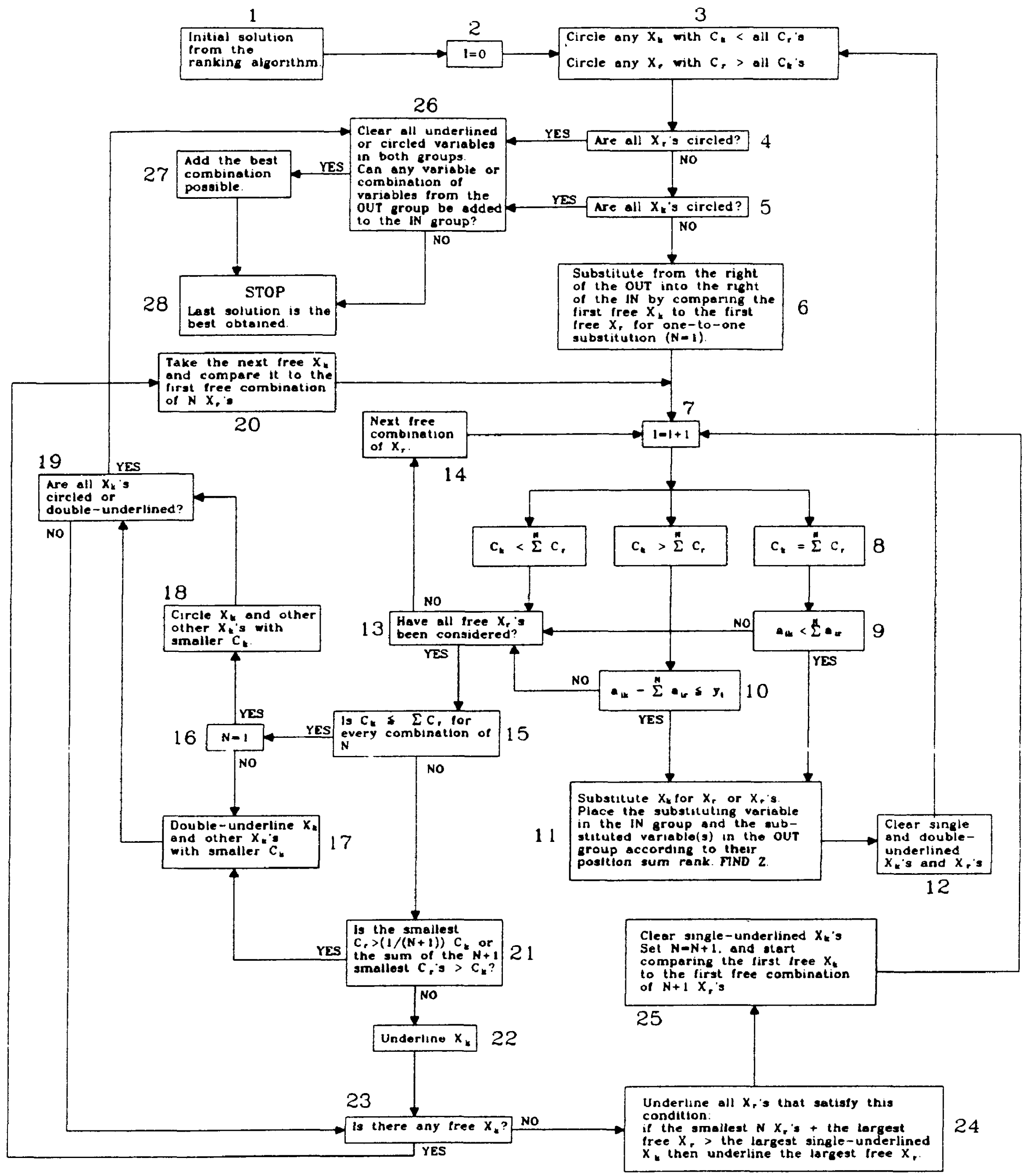

FIGURE 1

A BLOCK DIAGRAM OF THE ALGORITHM

The ranking scheme that is used to sort the variables facilitates quick substitution from the right of the OUT group to the right of the IN group and quick elimination of unpromising trials of possible substitution. By considering comparable sizes of constraint coefficients (a_{ij}), the possibility of fast substitution increases. However, the main driver of the method is the improvement in the objective function value. This is done by considering first the rightmost variables in the OUT group which commonly have large objective function coefficients (c_j). In order to substitute other variables, these variables will enter the IN group not only because they have large c_j's, but because they have advantageous a_{ij} sizes compared to their c_j's, and possibly they could produce or maintain enough slack (y_i) that will give room for subsequent substitutions or last minute addition of OUT group variables as described in step 7.

On the other hand, the leftmost free variables of the OUT group would most probably fail the c-test in trying to substitute ones on the right of the IN group but they would have a better chance as we move along towards the left of the IN group. This would translate into fewer computations per iteration and, in turn, execution of a large number of comparisons in a short time. The position sum ranking scheme that is produced by the first part of the algorithm is definitely more consistent and effective than any other ranking scheme to produce the largest feasible combination. This is even true in ill-behaved problems where more than one tight constraint have contradicting or less common variables in their largest feasible combinations. Moreover, the sorting of constraints from top to bottom in decreasing order of their tightness (represented by the HDC number) to be examined in that order for checking the feasibility of the current solution will help in sharply curtailing computation time.

The combinations of free IN group variables to be compared against free OUT group variables for possible substitution are examined in the order shown in the following example using a systematic combinations generator (All-combination generator).

EXAMPLE: Assume that there are four free IN group variables then, for substitution level 1 (i.e., one-to-one substitution) combinations are examined in the following order: (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0); for substitution level 2 (i.e., one-to-two substitution): (0,0,1,1), (0,1,0,1), (1,0,0,1), (0,1,1,0), (1,0,1,0), (1,1,0,0); for substitution level 3 (i.e., one-to-three substitution): (0,1,1,1), (1,0,1,1), (1,1,0,1), (1,1,1,0); and for substitution level 4 (i.e., one-to-four substitution): (1,1,1,1) where 1 indicates the variables in the generated combination being considered for substitution trials.

The algorithm makes use of this arrangement, the scheme described in the previous two paragraphs, and the small amount of calculations required per iteration to investigate a large number of promising combinations in a short time and to facilitate quick substitution and elimination of combinations. This combination generation scheme results in a more accurate algorithm than would be otherwise.

The permanent and temporary exclusion tests of variables from both groups (represented by steps 2, 3, 4, 5, and 6 of the substitution algorithm) will enhance the quick termination of the algorithm because combinations are only generated from free IN group variables and compared to free OUT group variables. These exclusion tests, though simple, are powerful as will be shown in the next section by examining the enumerated portion of possible combinations. The following example problem is solved step-by-step as indicated in the description of the ranking algorithm and the flow chart of the substitution algorithm.

EXAMPLE:

$$\begin{array}{llllllll} \text{Max } Z = & 100\ X_1 & +\ 500\ X_2 & +\ 600\ X_3 & +\ 2000\ X_4 & +\ 200\ X_5 & & \\ \text{St.} & 3\ X_1 & +\ 4\ X_2 & +\ 3\ X_3 & +\ 10\ X_4 & +\ 5\ X_5 & \le & 13 \\ & 2\ X_1 & +\ 4\ X_2 & +\ 2\ X_3 & +\ 3\ X_4 & +\ 2\ X_5 & \le & 10 \\ \hline \Sigma a_{ij} = & 5 & 8 & 5 & 13 & 7 & & \end{array}$$

The Ranking Algorithm

Step 1: For constraint 1: $X_3\ X_1\ X_2\ X_5\ X_4$

For constraint 2: $X_3\ X_1\ X_5\ X_4\ X_2$

Step 2: For constraint 1: HDC = 3

For constraint 2: HDC = 3

Step 3: Position sum for $X_1 = 4$, $X_2 = 8$, $X_3 = 2$, $X_4 = 9$, and $X_5 = 7$

The final order is: $X_3X_1X_5X_2X_4$ and HDC = 3

The IN group = $(X_3X_1X_5)$, the OUT group = (X_2X_4)

Step 4: Initial solution $(X_3X_1X_5) = 1$, $(X_2X_4) = 0$ is feasible, $Z = 900$.

The Substitution Algorithm

1. Initial solution = $(X_3X_1X_5) = 1$, $(X_2X_4) = 0$. $Z = 900$.
2. I=0
3. None
4. No
5. No

6. N=1, X_4 for X_5

7. I=1

8. $C_4 > C_5$

10. No

13. No

14. X_4 for X_1

7. I=2

8. $C_4 > C_1$

10. No

13. No

14. X_4 for X_3

7. I=3

8. $C_4 > C_3$

10. No

13. Yes

15. No

21. No

22. Underline X_4: $(X_3X_1X_5)=1$, $(X_2\underline{X_4})=0$, Z=900.

23. Yes

20. X_2 for X_5

7. I=4

8. $C_2 > C_5$

10. Yes

11. Substitution is executed.

Current solution: $(X_3\ X_1\ X_2)=1$, $(X_5\underline{X_4})=0$, Z=1200.

12. $(X_3X_1X_2)=1$, $(X_5X_4)=0$, Z=1200

3. None

4. No

5. No

6. N=1, X_4 for X_2

7. I=5

8. $C_4 > C_2$

10. No

13. No

14. X_4 for X_1

7. I=6

8. $C_4 > C_1$

10. No

13. No

14. X_4 for X_3

7. I=7

8. $C_4 > C_3$

10. No

13. Yes

15. No

21. No

22. Underline X_4: $(X_3X_1X_2)=1$, $(X_5\underline{X_4})=0$, Z=1200

23. Yes

20. X_5 for X_2

7. I=8

8. $C_5 < C_2$

13. No

14. X_5 for X_1

7. I=9

8. $C_5 > C_1$

10. Yes

11. Substitution is executed

Current solution: $(X_3X_5X_2)=1$, $(X_1\underline{X}_4)=0$, Z=1300.

12. $(X_3X_5X_2)=1$, $(X_1X_4)=0$, Z=1300.

3. Circle X_1: $(X_3X_5X_2)=1$, $(X_1X_4)=0$, Z=1300.

4. No

5. No

6. N=1, X_4 for X_2

7. I=10

8. $C_4 > C_2$

10. No

13. No

14. X_4 for X_5

7. I=11

8. $C_4 > C_5$

10. No

13. No
14. X_4 for X_3
7. I=12
8. $C_4 > C_3$
10. No
13. Yes
15. No
21. No
22. Underline X_4: $(X_3X_5X_2)=1$, $(X_1\underline{X_4})=0$, Z=1300
23. No
24. None
25. N=2: $(X_3X_5X_2)=1$, $(X_1X_4)=0$, Z=1300, X_4 for X_5 and X_2
7. I=13
8. $C_4 > C_5+C_2$
10. Yes
11. Substitution is executed.

    Current solution: $(X_3X_4)=1$, $(X_1X_5X_2)=0$, Z=2600
12. None
3. Circle X_2, X_3, X_4 and X_5: $(X_3X_4)=1$, $(X_1X_5X_2)=0$, Z=2600
4. Yes
26. No
28. $(X_3X_4)=1$, $(X_1X_5X_2)=0$, Z=2600 is the best solution obtained.

One has to think about other tests that emphasize the consideration of the matrix coefficients even if this is already partially considered by sorting according to the position sum criterion. However, one also has to consider the trade off between increasing the computation time per iteration as a result of coding these tests and saving time as a result of excluding more unpromising points. Some of these tests will be suggested in the last chapter for further extension of this work. In addition, the amount of computer memory required by this algorithm is insignificant compared to other methods. Besides the original data of the problem, only the last obtained solution, the position sum of the variables and an array for y_i, are present in the current memory state.

Since the algorithm executes the first feasible improvement encountered in the substitution pattern of the problem a good number of successive feasible solutions are produced. These feasible solutions can be very useful in decision making heavily influenced by politics not represented in the problem formulation. This is especially true when the optimum is not satisfactory to all parties involved but a solution very close to the optimum is. Also, the substitution pattern of the problem can reveal some insights and useful information from which unexpected observations can be deducted. A side benefit of this is that the best combinations of variables can be obtained in different number of combination levels between the largest combination allowed and the best obtained combination; i.e., best of 8, 7, 6, etc.

One may argue that it would be sufficient if only the greatest improvement encountered for any substituting variable is executed, or that instead of starting with a one-to-one substitution, one could start with higher substitution levels. Old greedy heuristic algorithms that use the first suggested procedure proved most of the time inaccurate and the second suggestion is unsystematic. Besides its side benefits mentioned above, the implementation of the first encountered feasible improvement in combination with the requirement of step 2 to go back to one-to-one substitution after each substitution in any level increased the accuracy of the algorithm in comparison to known heuristic algorithms in the literature. Finally, the last step of the algorithm was added in the later stages of developing this work and proved very powerful in closing the gap towards optimality for many problems.

Dr. Woolsey (1991) states that when solving IP problems one has to recognize obvious things first such as critical variables and constraints. Using the same idea I would argue that a good heuristic algorithm has to pick up these obvious things. The algorithm presented in this work, by means of the ranking procedure and the exclusion tests incorporated in the substitution procedure, takes care of much obvious information in the problem. However, this does not mean that it can completely eliminate the need, sometimes, for effective preprocessing procedures that experienced practioners consider very essential in solving large real-world problems.

3.4. FINITENESS AND NONREDUNDANCY OF THE ALGORITHM

As shown in the example above the comparison sequence is done according to a systematic generation of combinations of free IN group variables. This systematic way is preferred to random picking of variables in two ways. First, it enhances the chance of possible substitution because of the way the variables are sorted. Second, it enhances the accuracy of the algorithm by not skipping promising comparisons as much as possible. One may argue that a very large number of comparisons will be implemented which will hamper the speed of the algorithm and in turn its effectiveness. This is not true as it is shown below in the proof of the finiteness of the algorithm.

At any state during the substitution procedure there are n variables in the IN group and i variables in the OUT group, assuming that no variables are excluded temporarily or completely, then the maximum number of trials before the algorithm terminates with the same count in each group (no substitution takes place and the value of Z remains the same) is:

$$P_1 = i\sum_{r=1}^{n} \binom{n}{r} = i(2^n-1)$$

Now let's assume that substitution takes place at some level k_l where k is the number of free IN group variables to be substituted and l is the substitution sequence, m_l is the number of free OUT group variables that have been considered including the substituting variable, i_l and n_l are the number of free variables in both groups before the substitution sequence l, and, for simplicity, let's assume that substitution takes place to the last combination of free IN group variables generated in the level k; then, the maximum number of trials before the algorithm terminates, ignoring the notion of temporarily and completely excluded variables in both groups, is:

$$P_2 = \sum_{l=1}^{q}\left[i_l \sum_{r=1}^{k_l-1}\binom{n_l}{r} + m_l\binom{n_l}{k_l}\right] + L, \qquad l = 1, 2, \ldots, q$$

where

$$L = g \sum_{r=1}^{h}\binom{h}{r}$$

$i_l=i_{l-1}+k_{l-1}-1, \qquad l = 2, 3, \ldots, q$

$n_l=n_{l-1}-k_{l-1}+1, \qquad l = 2, 3, \ldots, q$

$g = i_q+k_q-1,$

$h = n_q-k_q-1,$

$i_1=i$, and $n_1=n$

The following numerical example illustrates the proof of the algorithm's finiteness.

EXAMPLE: Assume that $n=20$ and $i=10$, then

$$P_1 = 10\sum_{r=1}^{20}\binom{20}{r} = 10(20^{20}-1) = 10,485,750$$

comparisons (trials); however, for this example the maximum number of possible combinations is:

$$2^{n+i} = 2^{30} = 1,073,741,824$$

Then the portion of the enumerated combinations (points) is:

$$t = P_1/2^{n+i} = 0.01$$

Now let's assume that three substitutions take place according to the following pattern:

1st substitution => $k_1=1$, $m_1=3$

2nd substitution => $k_2=2$, $m_2=6$

3rd substitution => $k_3=3$, $m_3=5$

then

$$P_2 = 10\sum_{r=1}^{0}\binom{20}{r} + 3\binom{20}{1} + 10\sum_{r=1}^{2}\binom{20}{r} + 6\binom{20}{3} + \sum_{r=1}^{4}\binom{18}{r} + 5\binom{18}{5} + 16\sum_{r=1}^{14}\binom{14}{r}$$

$$= 0 + 60 + 2{,}100 + 6{,}840 + 48{,}444 + 42{,}840 + 262{,}128$$

$$= 362{,}412.$$

This is a 96.5% decrease from P_1. The portion of the enumerated combinations (points) for this example is:

$$t = P_2/2^{n+i} = 0.000338$$

This dramatic reduction occurred by executing only three substitutions and ignoring the notion of temporary and permanent exclusion of variables in both groups. However, for a problem of 30 variables, depending on the structure of the problem, a larger number of substitutions will take place and, by considering the notion of exclusion which decreases significantly the number of free variables, the reduction in the enumerated portion will be even more dramatic. For example, one of Petersen's problems with 28 variables was solved, 15 substitutions took place and the enumerated portion was t=0.000029. This would be even more dramatic for larger problems, but the enumerated portion could still be very large so that one would need either powerful computers to overcome this problem or clever algorithms that would enumerate even a smaller portion.

The comparison trials scheme of the algorithm is not redundant for two reasons. First, when substitution takes place the combinations of variables in both groups are different. So, if any OUT group variable is compared to the same combination of IN group variables again it will be under different conditions. Second, when there is no substitution, the substitution level is always increased by one after considering all free OUT group variables until every variable in the OUT group is circled or double underlined or every IN group variable is circled or underlined. Hence, every free variable in the OUT group is always compared to a larger combination of free IN group variables or more variables are excluded from both groups as the substitution level increases.

3.5. VERSIONS OF THE ALGORITHM

Different modifications of the algorithms are attempted to see if better results on some of the tested problems can be attained or faster termination of the algorithm can be achieved. Two major modifications and six minor ones are devised.

The first major modification is in step 1 of the substitution algorithm. Instead of starting from the right of the OUT group, the comparisons can be started from the left. The aim of this modification is to see if more substitutions could result in more accurate answers. The second major modification is that the generated combinations of free IN group variables for comparison to OUT group variables are generated differently. The generation process works as shown in the following example using the shift-combination generator. Using this procedure, a much smaller number of combinations is produced compared to that of the All-combination generator, especially as problems get larger.

EXAMPLE: Assume that there are four free IN group variables then, for substitution level 1 combinations are examined in the following order: (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0); for substitution level 2: (0,0,1,1), (0,1,1,0), (1,1,0,0); for substitution level 3: (0,1,1,1), (1,1,1,0); and for substitution level 4: (1,1,1,1).

The maximum number of combinations generated using this method, assuming no substitution and no exclusion of variables take place, is: $P = n(n+1)i/2$, where n is the number of free IN group variables and i is the number of free OUT group variables. The aim of this modification is to speed up the termination process and still maintain satisfactory accuracy of the algorithm. Another aim is to solve very large problems in a reasonably short time in order to obtain some insights about these problems that can be helpful when using the original algorithm, or to feed the obtained solution from this modification to the original algorithm.

The other minor modifications are options in the sorting procedure according to which the initial solution is obtained, and in the way variables in both groups are sorted during the substitution process. The different combinations of these modifications that are used are listed below.

1) Initial solution is obtained using position sum and both groups are sorted using position sum during substitution.
2) Initial solution is obtained using position sum and the IN group is sorted using position sum while the OUT group is sorted using objective function coefficients during substitution.
3) Initial solution is obtained using position sum and the IN group is sorted using objective function coefficients

while the OUT group is sorted using position sum during substitution.

4) Initial solution is obtained using objective function coefficients and the IN group is sorted using objective function coefficients while the OUT group is sorted using position sum during substitution.

5) Initial solution is obtained using position sum and both groups are sorted using objective function coefficients during substitution.

6) Initial solution is obtained using objective function coefficients and both groups are sorted using objective function coefficients during substitution.

Different experimental runs of these minor modifications in combination with the aforementioned two major modifications were attempted. Results of computational experience will be presented and analyzed in the next chapter. The original version of the algorithm will be referred to as version 111.

Chapter 4

COMPUTATIONAL EXPERIENCE

In order to verify the algorithm it was coded in turbo PASCAL and then was tested on 36 famous problems from the literature. The problems were run on an AT IBM-286 compatible 16 MHz PC. Most of the problems are of the capital budgeting type and are comprised of seven R&D project selection problems from Petersen (1967) ranging in size from 6 to 50 variables, two 60 variables by 30 constraints proposal mix optimizing problems from Senju and Toyoda (1968), ten fixed-charge problems by Haldi ranging in size from 11 to 30 variables and nine 10 variable single knapsack allocation problems from Trauth and Woolsey (1969), and eight multiple knapsack project selection problems from Weingartner and Ness (1967). The ten fixed-charge problems of Haldi were converted to 0-1 problems in order to be solved by our algorithm. The eight problems of Weingartner are originally two problems of 28 and 105 variables, respectively, for which the right hand sides were varied to generate these eight problems, six of 28 variables and two of 105 variables. All the problems are published in their respective papers.

Taha (1975) presented two major limitations with published computational experience. The first limitation is that used sample problems are not totally representative of the common types encountered in applications. The second limitation is that the sizes of these problems are relatively small, which does not show the effectiveness of tested algorithms. In order to overcome these two major limitations, the test problems used in this work are either real applications or have a real-world origin. The rest of the problems are recognized in the literature for their difficulty and approved suitability for testing algorithms such as the Haldi problems. Also, the problems range in size from 6 to 105 variables and from 1 to 30 constraints.

Computational experience results of the original algorithm and the best answers obtained are shown in Table 1. The original version of the algorithm was chosen for complete results tabulation because it is the most consistent version of the algorithm. In this table run, times include computations and sending every feasible solution found after every substitution to the output file. Sending only the first and final feasible solutions to the output file was attempted and it showed significant reduction in run times for some of the large test problems for which a large number of substitutions occurred. The problems were run with no limit on the substitution level, which means that the substitution level will increase until all the variables in either the IN or the OUT

TABLE 1

COMPUTATIONAL RESULTS OF THE ALGORITHM ON AN IBM-COMPATIBLE PC (16 MHz)

Problem	Size (nxm)	Answer obtained by version 111	% of optimum	Time (seconds)	Comments
Petersen#1	6x10	3800	100.0	0.60	
= 2	10x10	8707	100.0	0.77	
= 3	15x10	4015	100.0	1.05	
= 4	20x10	6120	100.0	1.48	
= 5	28x10	12400	100.0	2.36	
= 6	39x5	10583	99.67	114.14[1]	Other versions obtained 10588 (99.72%) in 80.0 sec.
= 7	50x5	16496	99.75	83.76[1]	Another version obtained 16519 (99.9%) in 87.2 sec.
Toyoda#1	60x30	7719	99.32[2]	6.31	Another version obtained the optimum in 5.6 sec.
= 2	60x30	8713	99.90[2]	7.52	
Allocation#1	10x1	50	100.0	0.60	
= 2	10x1	52	100.0	0.60	
= 3	10x1	55	96.49	0.60	Other versions obtained the optimum.
= 4	10x1	57	91.94	0.60	Other versions obtained the optimum.
= 5	10x1	67	100.0	0.60	
= 6	10x1	68	100.0	0.60	
= 7	10x1	70	100.0	0.60	
= 8	10x1	72	96.0	0.60	Other versions obtained the optimum.
= 9	10x1	85	100.0	0.60	
Fix-charge#1[3]	11x4	7	100.0	0.60	
= 2	12x4	8	100.0	0.60	
= 3	14x4	10	100.0	0.82	
= 4	12x4	8	100.0	0.66	
= 5	22x6	75	98.68	2.70	Other versions obtained the optimum in 2.80 sec.
= 6	23x6	104	98.11	7.96	Another version obtained the optimum in 4.67 sec.
= 7	22x4	75	98.68	2.69	Other versions obtained the optimum in 1.92 sec.
= 8	23x4	96	90.57	7.63	Another version obtained the optimum in 4.56 sec.
= 9	15x6	9	100.0	1.27	
= 10	30x10	14	82.35	1.49	Another version obtained 16 (94.12%) in 2.25 sec.
Weingartner#1	28x2	141278	100.0	1.53	
= 2	28x2	130883	100.0	1.70	
= 3	28x2	95677	100.0	1.59	
= 4	28x2	119088	99.79	2.41	Other versions obtained the optimum in 2-3.5 sec.
= 5	28x2	98631	99.83	1.87	Other versions obtained the optimum in2-3.4 sec.
= 6	28x2	130623	100.0	1.76	
= 7	105x2	1094945	99.95	39.41	Other versions obtained the optimum in 20-40 sec.
= 8	105x2	619766	99.27	71.19	Another version obtained 622376 (99.69%) in 31 sec.

1. Limit on number of iterations is used.
2. Optimal answers for these two problems are first obtained by Marsten and Morin (1977).
3. Alternative optimal answers to those reported by Trauth and Woosey (1969) are obtained for these problems.

groups are excluded. However, the substitution level could be limited to 3, 4, or 5, which will significantly decrease run times for large problems and affect the closeness of the final answer to the optimum. But, this could be a necessary practice for very large problems and still satisfactory results could be obtained. A sample of the output is presented in Appendix B.

The algorithm solved 31 out of the 36 test problems to optimality. The algorithm averaged more than 99.80% accuracy for all the problems where 35 problems had a lower limit of 99.70% of optimality and only one problem had more than 94.10% (this is the last fixed-charge problem).

The algorithm averaged less than 1.30 seconds for the first five problems of Petersen. The remaining two problems took an average of 80 seconds to reach more than 99.70% of optimality. These two problems have proven very difficult problems for exact and heuristic algorithms. Even though one cannot compare algorithms run on different machines from a run-time perspective, we can make comparisons on the basis of accuracy. The performance of our heuristic algorithm, from an accuracy point of view, on this set of problems is better than that of the three heuristics of Senju and Toyoda, Kochenberger et al., and Hillier which were examined by Zanakis (1977), and as good as the pivot and complement heuristic of Balas and

Martin (1980). The machines that were used in these studies are large computers, thus we cannot compare run times to that of our PC.

For the two Senju and Toyoda problems, the algorithm averaged less than 7 seconds and obtained an optimal answer for one and 99.90% of the optimal answer for the other. From an accuracy prospective, our algorithm outperformed the three heuristics in the Zanakis study.

The nine allocation problems were solved to optimality in an average of 0.60 seconds which is much better than the results obtained by Kochenberger et al. (1974), and comparable to that of exact cutting methods examined by Trauth and Woolsey (1969).

For the ten fixed-charge problems of Haldi, the algorithm obtained optimal answers for the first nine problems and solved the last one to 94.12% of optimality in an average run time of less than 2.7 seconds. These results are also much better than those reported in Kochenberger et al. (1974), and comparable to those in Trauth and Woolsey (1969) and Kim (1983). Moreover, alternative optimal answers were found for the first nine problems other than those reported by Trauth and Woolsey.

For the eight Weingartner and Ness problems, the algorithm averaged more than 99.95% accuracy. The first seven problems were solved optimally and the last one to 99.70% of optimality. The algorithm averaged 1.80 seconds for the first six problems and the best times reported for the last two problems were 20 and 31 seconds, respectively. These results are better than those reported in Zanakis' study and impressive compared to the exact dynamic programming algorithms by Weingartner and Ness (1969). A composite summary of these results are shown in Table 2.

As was mentioned in the previous chapter, the shift-combination generation scheme was devised to speed up the examination process of possible substitutions while maintaining a satisfactory level of accuracy. Although this process is less accurate than the all-combination generation scheme which examines a much larger number of possible substitutions, the algorithm using this option has produced impressive results. Only 27 problems were used to test the algorithm under this option, excluding the nine allocation problems because they did not show any reduction in run time or accuracy. Computational experience results are shown in Table 3. The algorithm using our least accurate option averaged more than 98.10% of accuracy. Out of the 27 test problems, 14 were solved optimally, 17 with more than 99.90% accuracy, 22 with more than 99.0%, 24 with more than 96.0%, 26 with more than 87%, and only one problem with 82.35% of the optimal.

TABLE 2

A COMPOSITE SUMMARY OF COMPUTATIONAL RESULTS OF THE ALGORITHM

Problems	average % of optimum		optimal problems out of total	
	version 111	best version	version 111	best version
Petersen	99.92	99.94	5/7	5/7
Toyoda	99.61	99.95	0/2	1/2
Allocation	98.27	100.00	6/9	9/9
Fixed-charge	96.84	99.41	5/10	9/10
Weingartner	99.86	99.96	4/8	7/8
Composite	98.62	99.81	20/36	31/36

TABLE 3

COMPUTATIONAL RESULTS OF THE ALGORITHM USING
THE SHIFT-COMBINATION GENERATION SCHEME
ON AN IBM-COMPATIBLE PC (16 MHz)

Problem	Size	Best answer by any version	% of optimal	Time (sec.)
Petersen#1	6x10	3800	100.0	0.54
= 2	10x10	8707	100.0	0.71
= 3	15x10	4015	100.0	1.04
= 4	20x10	6120	100.0	1.21
= 5	28x10	12390	99.92	1.97
= 6	39x5	10588	99.72	2.64
= 7	50x5	16397	99.15	4.51
Toyoda#1	60x30	7728	99.43	5.87
= 2	60x30	8713	99.90	7.31
Fix-charge#1	11x4	7	100.0	0.60
= 2	12x4	8	100.0	0.60
= 3	14x4	10	100.0	0.72
= 4	12x4	8	100.0	0.66
= 5	22x6	75	98.68	1.21
= 6	23x6	93	87.74	1.10
= 7	22x4	73	96.05	0.99
= 8	23x4	93	87.74	0.99
= 9	15x6	9	100.0	0.72
= 10	30x10	14	82.35	1.48
Weingartner#1	28x2	141278	100.0	1.21
= 2	28x2	130883	100.0	1.43
= 3	28x2	94908	99.20	1.37
= 4	28x2	119337	100.0	1.15
= 5	28x2	98796	100.0	1.43
= 6	28x2	130623	100.0	1.54
= 7	105x2	1094945	99.95	10.17
= 8	105x2	619413	99.20	22.69

The algorithm produced incredible reductions in running times, especially with large problems.

Under this option the algorithm solved the seven Petersen problems with an average of more than 99.80% of optimality. It averaged less than 1.10 seconds for the first five problems compared to an average of 1.30 seconds under the all-combination version. However, for the two largest Petersen problems, the algorithm averaged less than 3.60 seconds compared to 80 seconds of the more accurate version, losing an average of only less than 0.40% of accuracy. For the two Senju and Toyoda problems there were not significant reductions in accuracy and time; the algorithm averaged around 6.60 seconds and 99.67% accuracy. The algorithm also averaged more than 95.0% of the optimal answer for the ten fixed-charge problems in less than a second compared to 2.70 seconds of the original version, losing only an average of 4.0% in accuracy. Regarding the eight problems of Weingartner and Ness, the algorithm averaged 99.80% of optimality in less than 1.40 seconds for the first six problems and less than 16.50 seconds for the two largest problems, losing only an average of 0.16% accuracy.

These results, though of our least accurate option, are better than the results of other heuristics discussed in the above mentioned literature and support our claim in the previous chapter that running the algorithm under this option can solve very large problems in a short time and still produce good results. Also, the final solution of this version can be used as an initial solution for exact algorithms. Dr. Woolsey once said that "the optimal answer is as good as the accuracy of the data." That is why heuristics are so popular in this world of business and politics, especially from decision makers.

Now to validate the speed of our algorithm, it has to be compared to other algorithms. Our algorithm, the Balas algorithm, and LINDO, a commercial package which uses LP as a first step to provide an upper bound solution from which a branch-and-bound scheme searches for the optimal answer, have been tested using 29 out of the 36 test problems on the same machine used above in order to make an objective comparison between the three methods. The problems represent all the test problem groups and range in size from 6 to 105 variables and from 1 to 30 constraints. There was no need to include all the nine allocation problems because they ran for the same time. The Balas method was first run using a BASIC code obtained from Kilinc (1989) but it was much slower than our algorithm for every problem tested. This could be attributed partly to the different computer languages used to code the two algorithms. In order to eliminate the problem of code dependency as much as possible, especially from a programming language point of view, a PASCAL code of Balas's method was obtained from Syslo et al. (1983) and then was modified to be compatible with turbo PASCAL, the same language used to code our algorithm. LINDO was tested on only 10 problems because of the inconvenience of entering the data. However, these ten problems are representative enough to give a valid idea about the performance of LINDO as shown in the comparison table.

As shown in Table 4, our algorithm compares favorably to the other methods. The Balas method was faster than our algorithm with small problems. However, for problems larger than 15 variables, our algorithm was much faster in almost every case except for the two largest Petersen problems where limits on the number of iterations were used and answers with more than 99.70% of optimality were obtained. This is true with the fact that run time for our algorithm includes sending every feasible solution found by the algorithm to the output file which is not the case for the Balas code where the output file contains the final solution only. This was found to be a significant time for some large problems.

The algorithm also outperformed LINDO in 8 out of the 10 test problems and for the other two problems, it can be very long before LINDO achieves optimality. Since LINDO uses linear programming as a first step, this will even hamper its speed in larger problems and will fall short of achieving the

TABLE 4

COMPUTATIONAL RESULTS OF THREE METHODS ON
AN IBM-COMPATIBLE PC (16 MHz)
(TIME IN SECONDS)

Problem	Balas		LINDO		Algorithm	
	time	% of opt.	time	% of opt.	time	% of opt.
Petersen#1	0.05	100.0	5.22	100.0	0.60	100.0
= 2	0.06	100.0	9.48	100.0	0.77	100.0
= 3	0.27	100.0	29.82	100.0	1.05	100.0
= 4	4.34	100.0	35.48	100.0	1.48	100.0
= 5	65.15	100.0	44.64	100.0	2.36	100.0
= 6	219.37	100.0	40.20[1]	99.88	80.00[2]	99.72
= 7	7975.90	100.0	56.52[1]	99.46	87.20[2]	99.89
Allocation#1	0.05	100.0	3.36	100.0	0.60	100.0
= 2	0.05	100.0	4.67	100.0	0.60	100.0
Weingartner#1	1.98	100.0	13.23[1]	99.65	1.53	100.0
= 2	4.11	100.0	-	-	1.70	100.0
= 3	3.79	100.0	-	-	1.59	100.0
= 4	0.38	100.0	-	-	1.98	100.0
= 5	6.20	100.0	-	-	1.81	100.0
= 6	4.72	100.0	-	-	1.76	100.0
= 7	*	-	-	-	20.11	100.0
= 8	*	-	-	-	30.92	99.69
Fix-charge#1	0.22	100.0	-	-	0.60	100.0
= 2	0.22	100.0	-	-	0.60	100.0
= 3	0.44	100.0	-	-	0.82	100.0
= 4	0.16	100.0	-	-	0.66	100.0
= 5	56.41	100.0	-	-	2.80	100.0
= 6	158.68	100.0	-	-	4.67	100.0
= 7	51.58	100.0	-	-	1.92	100.0
= 8	145.61	100.0	-	-	4.56	100.0
= 9	0.44	100.0	-	-	1.27	100.0
= 10	117.63	100.0	-	-	2.25	94.12
Toyoda#1	**	-	-	-	5.60	100.0
= 2	**	-	-	-	7.47	99.90

1. Batch command is used for limits on iterations.
2. Limit on number of iterations is used.
* Code ran for more than 7 hours then was terminated.
** Code ran for more than 2 hours then was terminated.

run times of our algorithm. This is especially true for the two large Senju and Toyoda problems and for the two largest problems of Weingartner and Ness. LINDO spent almost 45 seconds to solve the fourth largest Petersen problem (20 variables), while our algorithm solved the largest four test problems (60, and 105 variables) in an average of 7 and 30 seconds, respectively.

Concerning the modification of starting substitution trials from the left of the OUT group instead from the right, it was observed that it produced longer times most of the time and more accurate results sometimes which coincides with our expectations mentioned in the previous chapter. Also, the algorithm has been tested on some combinatorial and set-covering problems and has produced good results. Finally, our experience with the algorithm suggests that most of the time more than 99.0% of the optimal answer is found in a short time and the rest of the time is spent on improving this solution. This would imply that a close to optimal solution could be obtained if the algorithm is terminated early. Moreover, it may suggest the need for more exclusion tests to enhance the termination process of the algorithm though most of the reported running times are satisfactory, considering the use of a PC.

Chapter 5

CONCLUSION AND SUGGESTIONS

Most of the known IP heuristics are specially designed to solve certain types of models or problems. The lack of new original heuristics that can be as general as possible since the pivot and complement algorithm of Balas and Martin (1980) encouraged the author to take this towards developing an efficient heuristic algorithm that can be easily understood by decision makers, academicians, practioners, and students of management science and operations research.

In this work a new effective heuristic algorithm that can solve a class of economic models formulated as 0-1 integer programs is presented. A wide range of capital budgeting problems of up to 105 variables were solved in less than 1.50 minutes on a PC using the most exhaustive version of the algorithm to an average of more than 99.80% of optimality, and in less than 1/4 of a minute using the least exhaustive version of the algorithm to an average of more than 98.50% of optimality. The algorithm also outperformed well-known algorithms, heuristics, and commercial packages using even less accurate versions of the algorithm sometimes. It provides the possibility of solving large problems of more than 100 variables to optimal or near optimal solutions on a PC in a very short time, especially using the fastest versions of the algorithm. Our work is a major step forward in that regard since, to the knowledge of this author, these large problems have never been solved on a PC. In summary, the algorithm has produced impressive results from the perspective of accuracy and running time when compared to existing known methods.

The algorithm differs from other known algorithms in a number of ways. It starts with the largest combination of variables allowed by the problem as an initial solution. This is done instead of starting with an all zero or one solution such as the methods of Balas (1965) and Lawler and Bell (1966) or those solutions obtained from maximum pay-off or other criteria. Also, this method uses a controlled search pattern that is substantially unaffected by individual problem structure. Using the ranking scheme of the variables, the algorithm forces the search towards the first most probable substitution or improvement in the value of the objective function with the minimum amount of calculations per iteration. The algorithm successively accumulates encountered feasible improvements without any limits or bounds on the size of the improvement. The termination process is not vague as in some of the algorithms, and one can follow the process towards the termination point easily. Finally, the search process of our method is a forward search scheme in the sense that there is no backtracking as in Balas's method. Even though a backtracking procedure can be very essential to ensure optimality, it proves to be very costly in execution time.

Results of computational experience of the algorithm support our claim that the algorithm would be very appealing to decision makers and practitioners who are interested in solving large problems repetitively. Also our algorithm estimates that most of the time, 99.0% of the optimal answer is found in a very short time and that the rest of the time is spent on improving upon this solution. Thus, excellent results still can be obtained if an early termination of the algorithm is desired or imposed. Therefore, the algorithm proves to be very useful and can be used alone or in association with other heuristic or exact algorithms. Although the algorithm has a number of implicit preprocessing considerations, the author believes that running the algorithm with more powerful preprocessing procedures would make it more effective.

There are areas of improvement in this work to enhance the termination process and to increase the accuracy of the algorithm. This can be done by constructing tests to exclude the examination of more unnecessary combinations (points) and finding new ways of discovering unexamined combinations. Some of these tests and procedures are very complicated and their coding could hamper the speed of the algorithm. However, the trade-offs generated from saving time, especially on large problems, by excluding more points and from increasing the accuracy of the algorithm need to be considered. Some of the suggestions in that regard are:

1) Although the algorithm eliminates variables permanently and temporarily from being considered in substitution trials at the start and throughout the procedure, more criteria are needed that will eliminate more variables without affecting the accuracy of the algorithm.
2) Incorporating more of the coefficients of the a-matrix in exclusion tests in order to determine early that an OUT group variable will not be able to substitute any variable or combination of variables from the IN group; i.e., how far down the IN group an OUT group variable should be involved in comparison trials.
3) Developing more criteria to accelerate the termination process of the algorithm especially when the algorithm reaches the mode of checking on the final obtained solution by going to higher and higher substitution levels until all the variables in either group are excluded.

4) Examining the steps that require the need to free all underlined OUT group variables or the starting of comparisons again from the right of the OUT group after each substitution. This could be done by monitoring the size of the current y vector; if its size after substitution gets larger, then previously examined failing variables need to be examined again before proceeding further down to the left of the OUT group, otherwise not. This suggestion could work and it is easy to implement, however, it needs to be tested for accuracy.

5) In order to increase the accuracy of the algorithm, the substitution procedure can be restarted after the last step for a limited number of levels (for example, 3). This can be an option only for those problems where the last step of the algorithm has improved the final solution.

6) Examining the need for group-to-one or more substitution after the last step or in other stages of the algorithm which may help in obtaining better solutions. However, this procedure is very complicated, time consuming, and its marginal benefit is very small.

7) Extending this work to deal with other types of models and to test the algorithm on larger problems, perhaps using mainframe computers or work stations.

REFERENCES CITED

Balas, E. 1965. "An Additive Algorithm for Solving Linear Programs with Variables." Operations Research, Vol. 13, pp. 517-546.

Balas, E., and C. Martin, 1980. "Pivot and Complement: A Heuristic for 0-1 Programming." Management Science, Vol. 26, pp. 86-96.

Balas, E., and E. Zemel, 1980. "An Algorithm for Large Zero-One Knapsack Problems." Operations Research, Vol. 28, pp. 1130-1154.

Balinski, M. L. 1965. "Integer Programming: Methods, Uses, Comportation." Management Science, Vol. 12, pp. 253-313.

Cabot, A. V. 1970. "An Enumeration Algorithm for Knapsack Problems." Operations Research, Vol. 18, pp. 306-311.

Crowder, H., E. L. Johnson, and M. Padberg, 1983."Solving Large-Scale Zero-one Linear Programming Problems." Operations Research, Vol. 31, pp. 803-834.

Dantzig, G. B., D. R. Fulkerson, and S. M. Johnson, 1954. "Solution of a Large Scale Traveling Salesman Problem." Operations Research, Vol. 2, pp.393-410.

Fayard, D., and G. Plateau, 1975. "Resolution of the 0-1 Knapsack Problem: Comparison of Methods." Mathematical Programming, Vol. 8, pp. 272-307.

Fleischmann, B. 1967. "Computational Experience With the Algorithm of Balas." Operations Research, Vol. 15, pp. 153-155.

Freeman, R. J. 1966. "Computational Experience With a 'Balas-ian' Integer Programming Algorithm." Operations Research, Vol. 14, pp. 935-941.

Geoffrion, A. M. 1967. "Integer Programming by Implicit Enumeration and Balas' Method." SIAM Review, Vol. 9, pp. 178-190.

______________. 1969. "An Improved Implicit Enumeration Approach for Integer Programming." Vol. 17, pp. 437-454.

Geoffrion, A. M., and R. E. Marsten, 1972. "Integer Programming Algorithms: A Framework and State of-the-Art Survey." Management Science, Vol. 18, pp. 465-491.

Gilmore, P. C., and R. E. Gomory, 1961. " A Linear Programming Approach to the Cutting Stock Problem." Operations Research, Vol. 9, pp. 849-859.

Glover, F. 1965. "A Multiphase-Dual Algorithm for the Zero-One Integer Programming Problem." Operations Research, Vol. 13, pp. 879-919.

__________. 1968. "Surrogate Constraints." Operations Research, Vol. 15, pp. 741-749.

Greenberg, H. 1971. Integer Programming. Academic Press, New York, NY.

Greenberg, H., and R. Hegerich, 1970. "A Branch Search Algorithm for the Knapsack Problem." Management Science, Vol. 16, pp. 327-332.

Hillier, F. S. 1969. "Efficient Heuristic Procedures for Integer Linear Programming with an Interior." Operations Research, Vol. 17, pp. 600-637.

Ingargiola, G. P., and J. F. Korsh, 1973. "Reduction Algorithm for Zero-One Single Knapsack Problems." Management Science, Vol. 20, pp. 460-463.

Johnson, E. L., M. M. Kostreva, and U. H. Suhl, 1985. "Solving 0-1 Integer Programming Problems Arising from Large Scale Planning Models, Operations Research, Vol. 33, pp. 803-819.

Kilinc, T. 1989. "An Improved Algorithm for the Selection of Constrained Investment Projects." Master's Thesis, Colorado School of Mines, Golden, Colo.

Kim, T. P. 1983. "A Reformulation Algorithm for a Class of Capital Investment Problems." Ph.D. Dissertation, Colorado School of Mines, Golden, Colo,.

Kochenberger, G. A., B. A. McCarl, and F. P. Wyman, 1974. "A Heuristic for General Integer Programming." Decision Sciences, Vol. 5, pp. 36-44.

Kuehn, A. A., and M. J. Hamburger, 1963. "A Heuristic Program for Locating Warehouses." Management Science, Vol. 9, pp. 643-666.

Land , A. H., and A. G. Doig, 1960. "An Automatic Method for Solving Discrete Programming." Econometrica, Vol. 28, pp. 497-520.

Lawler, E. L., and M. D. Bell, 1966. "A Method for Solving Discrete Optimization Problems." Operations Research, Vol. 14, 1966, pp. 1098-1112.

Lawler, E. L., and D. E. Wood, 1966. "Branch-And-Bound Methods: A Survey." Operations Research, Vol. 14, pp. 699-719.

Lemke, C. E., and K. Spielberg, 1967. "Direct Search Algorithms for Zero-One and Mixed-Integer Programming." Operations Research, Vol. 15, pp. 892-913.

Little, J. D. C., et al. 1963. " An Algorithm for the Traveling Salesman Problem." Operations Research, Vol. 11, pp. 979-989.

Lorie, J. H., and L. J. Savage, 1955. "Three Problems in Rationing Capital." Journal of Business, Vol. 28, pp. 229-239.

Manne, A. J. 1964. "Plant Location Under Economies-of-Scale, Decentralization and Computation." Management Science, Vol. 11, pp. 213-235.

Marsten, R. E., and T. L. Morin, 1977. "Optimal Solutions Found for Senju and Toyoda's 0/1 Integer Programming Problems." Management Science, Vol. 23, pp. 1364-1365.

Martello, S., and P. Toth, 1988. "A New Algorithm for the 0-1 Knapsack Problem." Managerment Science, Vol. 34, pp. 633-644.

Nauss, R. M. 1976. "An Efficient Algorithm for the 0-1 Knapsack Problem." Management Science, Vol. 23, pp. 27-31.

Nemhauser, G.L., and L.A. Wolsey, 1988. Integer and Combinatorial Optimization. John Wiley & Sons, Inc.

Petersen, C. C. 1967 "Computational Experience With Variants of the Balas Algorithm Applied to the Selection or R&D Projects." Management Science, Vol. 13, pp. 736-750.

Pierce, J. F. 1968. "Application of Combinatorial Programming to a Class of All-Zero-One Integer Programming Problems." Management Science, Vol. 15, pp. 191-209.

Plane, D. R., and C. McMillan, Jr. 1971. Discrete Optimization. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Rieter, S., and G. Sherman, 1965. "Discrete Optimizing." SIAM Journal, Vol. 13, 1965, pp. 864-889.

Schrage, L. 1989. Linear, Integer, and Quadratic Programming with LINDO. The Scientific Press, South San Francisco, CA.

Senju, S., and Y. Toyoda, 1968. "An Approach to Linear Programming with 0-1 Variables." Management Science, Vol. 15, pp. B-196-B-207.

Syslo, M. M., N. Deo, and J. S. Kowalik, 1983. Discrete Optimization Algorithms. Prentice Hall, INc., Englewood Cliffs, NJ.

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

Toyoda, Y. 1975. "A Simplified Algorithm for Obtaining Approximate Solutions to Zero-One Programming Problems." Management Science, Vol. 21, pp. 1417-1427.

Trauth, C. A., and R. E. D. Woolsey, 1969. "Integer Linear Programming: A Study in Computational Efficiency." Management Science, Vol. 15, pp. 481-493.

Weingartner, H. M. 1963. Mathematical Programming and the Analysis of Capital Budgeting Problems. Prentice-Hall, Inc., Englewood Cliffs, NJ.

_______________. 1966. "Capital Budgeting of Interrelated Projects: Survey and Synthesis." Management Science, Vol. 12, pp. 485-516.

Weingartner, H. M., and D. N. Ness, 1967. "Methods for the Solution of the Multi-Dimensional 0/1 Knapsack Problem." Operations Research, Vol. 15, pp. 83-103.

Woolsey, R. E. D. 1972. "A Candle to Saint Jude, OR Four Real World Applications of Integer Programming." Interfaces, Vol. 2, pp. 20-27.

__________. 1988. Integer Programming Course Handouts. Colorado School of Mines, Golden, Colo.

__________. 1991. Private Communication.

Zanakis, S. H. 1977. "Heuristic 0-1 Linear Programming: An Experimental Comparison of Three Methods." Management Science, Vol. 24, pp. 91-104.

Zionts, S. 1974. Linear and Integer Programming. Prentice-Hall, Inc., Englewood Cliffs, NJ.

APPENDIX A

EXAMPLE 1

MAX Z = 10 X1 + 7 X2 + 20 X3

st. 4 X1 + 3 X2 + 10 X3 <= 10

2 X1 + 2 X2 + 5 X3 <= 9

6 5 15

Ranking Order

Constraint 1: X2 X1 X3 , HDC=2
Constraint 2: X2 X1 X3 , HDC=3

Final Ranking Order: X2 X1 X3 , HDC=2

I	IN Group	OUT Group	Substitution	C-Test	A-Test	Z	Comments
0	(X2,X1)	(X3)	none	-	-	17	Initial solution. No variable can be eliminated from substitution trials and set N=1.
1	(X2,X3)	(X1)	X3 for X1	yes	no	-	All IN variables considered? no
2	(X1,X3)	(X2)	X3 for X2	yes	no	-	All IN variables considered? yes Is the sum of the two smallest C values of the IN variables > C3? no Single-underline X3.
	(X2,X1)	(<u>X3</u>)					Are there any free OUT variable? no Clear single-underlined OUT variables. Set N=N+1.
3	(X3)	(X2,X1)	X3 for X2, X1	yes	yes	20	Can any variables from both groups be eliminated from substitution trials? yes.
	(X3)	(X2,X1)				20	Are all IN variables circled? yes. STOP. OR Are all OUT variables circled? yes. STOP.

EXAMPLE 2

MAX Z = 100 X1 + 500 X2 + 600 X3 + 2000 X4 + 200 X5

ST.	3 X1 +	4 X2 +	3 X3 +	10 X4 +	5 X5 <= 13
	2 X1 +	4 X2 +	2 X3 +	3 X4 +	2 X5 <= 10
	5	8	5	13	7

Ranking Order

Constraint 1: X3 X1 X2 X5 X4 , HDC=3
Constraint 2: X3 X1 X5 X4 X2 , HDC=4

Final Ranking Order: X3 X1 X5 X2 X4 , HDC=3

I	IN Group	OUT Group	Substitution	C-Test	A-Test	Z	Comments
0	(X3,X1,X5)	(X2,X4)	none	-	-	900	Initial solution. Set N=1 Can any variable be eliminated from both groups? NO. Begin with the rightmost OUT variable.
1	(X3,X1,X4)	(X5,X2)	X4 for X5	yes	no	-	Next IN group variable.
2	(X3,X5,X4)	(X1,X2)	X4 for X1	yes	no	-	Next IN group variable.
3	(X1,X5,X4)	(X3,X2)	X4 for X3	yes	no	-	Have all free IN variables been considered? YES. Underline X4.
	(X3,X1,X5)	(X2,X4)	-	-	-	900	Next OUT group variable.
4	(X3,X1,X2)	(X5,X4)	X2 for X5	yes	yes	1200	Clear any underlined IN or OUT variables. Can any variable be eliminated? No. Set N=1
	(X3,X1,X2)	(X5,X4)	-	-	-	1200	First free OUT group variable.
5	(X3,X1,X4)	(X5,X2)	X4 for X2	yes	no	-	Next IN group variable.
6	(X3,X2,X4)	(X1,X5)	X4 for X1	yes	no	-	Next IN group variable.
7	(X1,X2,X4)	(X3,X5)	X4 for X3	yes	no	-	Underline X4.
	(X3,X1,X2)	(X5,X4)	-	-	-	1200	Next OUT group variable.
8	(X3,X1,X5)	(X2,X4)	X5 for X2	no	-	-	Next IN group variable.
9	(X3,X5,X2)	(X1,X4)	X5 for X1	yes	yes	1300	Clear underlined variables. Can any variables be eliminated from both groups? YES. Set N=1
	(X3,X5,X2)	(X1,X4)	-	-	-	1300	First free OUT group variable.
10	(X3,X5,X4)	(X1,X2)	X4 for X2	yes	no	-	Next IN group variable.
11	(X3,X2,X4)	(X1,X5)	X4 for X5	yes	no		Next IN group variable.
12	(X5,X2,X4)	(X1,X3)	X4 for X3	yes	no	-	Underline X4.
	(X3,X5,X2)	(X1,X4)	-	-	-	1300	No OUT group variable is free. Can any IN variable be eliminated? NO. Set N=N+1 (substitution level 2). Clear any single-underlined variables.
	(X3,X5,X2)	(X1,X4)	-	-	-	1300	First free OUT group variable.
13	(X3,X4)	(X1,X5,X2)	X4 for X5,X2	yes	yes	2600	Clear any underlined variables. Can any variables be eliminated? YES.
	(X3)(X4)	(X1)(X5)(X2)			-	2600	All IN variables circled. STOP OR All OUT variable circled. STOP

APPENDIX B

Petersen #1

100	600	1200	2400	500	2000	
8	12	13	64	22	41	<= 80
3	6	4	18	6	4	<= 20
5	10	8	32	6	12	<= 36
5	13	8	42	6	20	<= 44
5	13	8	48	6	20	<= 48
8	12	13	75	22	41	<= 96
3	2	4	8	8	4	<= 24
0	0	0	0	8	0	<= 10
3	0	4	0	8	0	<= 18
3	2	4	0	8	4	<= 22
43	70	66	287	100	146	

1	2	3	4	5	6	HDC
1	2	3	6	4	5	4
1	4	2	6	5	3	4
1	4	3	6	2	5	4
1	4	3	6	2	5	4
1	4	3	6	2	5	4
1	2	3	6	4	5	5
2	1	3	6	5	4	5
1	3	2	5	6	4	6
4	1	5	3	6	2	6
3	2	4	1	6	5	6
16	27	31	51	42	43	

Combination Generator used : ALL COMBINATIONS.
IN group sorting parameter : RANKS SUM.
OUT group sorting parameter : RANKS SUM.
IN group variables sorted in INCREASING order.
OUT group variables sorted in INCREASING order.

Inactive In group var(s) = 0
Inactive out group var(s) = 0
In group count = 4
Out group count = 2

(1,2,3,5) <--> (6,4)

Z = 2400
Iteration count = 0

```
Substituting Var            =  ( 6 )
Substituted Var(s)          =  ( 3,5 )
Inactive In group var(s)    =  0
Inactive out group var(s) =  0
In group count    =  3
Out group count   =  3

( 1,2,6 ) <--> ( 3,5,4 )

Z   =   2700
Iteration count   = 15
------------------------------------------------------
Substituting Var            =  ( 5 )
Substituted Var(s)          =  ( 1 )
Inactive In group var(s)    =  0
Inactive out group var(s) =  ( 1 )
In group count    =  3
Out group count   =  3

( 2,5,6 ) <--> ( 1,3,4 )

Z   =   3100
Iteration count   = 21
-------------------------------------------------------
Substituting Var            =  ( 3 )
Substituted Var(s)          =  ( 5 )
Inactive In group var(s)    =  0
Inactive out group var(s) =  ( 1,5 )
In group count    =  3
Out group count   =  3

( 2,3,6 ) <--> ( 1,5,4 )

Z   =   3800
Iteration count   = 26
------------------------------------------------------

Final iteration count   = 33

Calculation time   =    00:00:00.66
```

Petersen #2

600	311	1800	3850	19	190	882	4200	403	327	
20	5	100	200	2	4	60	150	80	40	<= 450
20	7	130	280	2	8	110	210	100	40	<= 540
65	14	80	220	6	10	50	180	30	50	<= 480
60	3	50	100	4	2	20	40	6	12	<= 200
60	8	70	200	4	6	40	70	16	20	<= 360
60	13	70	250	4	10	60	90	20	24	<= 440
60	13	70	280	4	10	70	105	22	28	<= 480
5	2	20	100	2	5	10	60	0	0	<= 200
45	14	80	180	6	10	40	100	20	0	<= 360
55	14	80	200	6	10	50	140	30	40	<= 440
450	93	750	2010	40	75	510	1145	324	254	

1	2	3	4	5	6	7	8	9	10	HDC
4	3	8	10	1	2	6	9	7	5	8
4	2	8	10	1	3	7	9	6	5	8
7	3	8	10	1	2	6	9	4	5	8
9	2	8	10	3	1	6	7	4	5	9
7	3	8	10	1	2	6	9	4	5	9
6	3	8	10	1	2	7	9	4	5	9
6	3	8	10	1	2	7	9	4	5	9
6	4	8	10	3	5	7	9	2	1	9
7	4	8	10	2	3	6	9	5	1	9
7	3	8	10	1	2	6	9	4	5	9
63	30	80	100	15	24	64	88	44	42	

```
Combination Generator used :  ALL COMBINATIONS..
IN group sorting parameter :  RANKS SUM.
OUT group sorting parameter :  RANKS SUM.
IN group variables sorted in INCREASING order.
OUT group variables sorted in INCREASING order.

---------------------------------------------------
 Inactive In group var(s)  =  0
 Inactive out group var(s) =  0
 In group count    =  8
 Out group count   =  2

 ( 5,6,2,10,9,1,7,3 ) <--> ( 8,4 )

 Z  =  4532
 Iteration count  = 0
---------------------------------------------------
```

```
 Substituting Var           =  ( 8 )
 Substituted Var(s)         =  ( 3 )
 Inactive In group var(s)   =  ( 8 )
 Inactive out group var(s)  =  0.
 In group count    =  8
 Out group count   =  2

 ( 5,6,2,10,9,1,7,8 ) <--> ( 3,4 )

 Z  =  6932
 Iteration count  = 9
---------------------------------------------------
 Substituting Var           =  ( 3 )
 Substituted Var(s)         =  ( 7 )
 Inactive In group var(s)   =  ( 8 )
 Inactive out group var(s)  =  0
 In group count    =  8
 Out group count   =  2

 ( 5,6,2,10,9,1,3,8 ) <--> ( 7,4 )

 Z  =  7850
 Iteration count  = 17
---------------------------------------------------
 Substituting Var           =  ( 7 )
 Substituted Var(s)         =  ( 9 )
 Inactive In group var(s)   =  ( 8 )
 Inactive out group var(s)  =  0
 In group count    =  8
 Out group count   =  2

 ( 5,6,2,10,1,7,3,8 ) <--> ( 9,4 )

 Z  =  8329
 Iteration count  = 27
---------------------------------------------------
 Substituting Var           =  ( 4 )
 Substituted Var(s)         =  ( 10,1,7,3 )
 Inactive In group var(s)   =  ( 8,4 )
 Inactive out group var(s)  =  0
 In group count    =  5
 Out group count   =  5

 ( 5,6,2,8,4 ) <--> ( 10,9,1,7,3 )

 Z  =  8570
 Iteration count  = 119
---------------------------------------------------
```

```
Substituting Var           =  ( 10 )
Substituted Var(s)         =  ( 2 )
Inactive In group var(s)   =  ( 8,4 )
Inactive out group var(s) =  0
In group count   =  5
Out group count  =  5

( 5,6,10,8,4 ) <--> ( 2,9,1,7,3 )

Z  =  8586
Iteration count  = 132
-----------------------------------------------------
Substituting Var           =  ( 2 )
Substituted Var(s)         =  ( 6 )
Inactive In group var(s)   =  ( 8,4 )
Inactive out group var(s) =  0
In group count   =  5
Out group count  =  5

( 5,2,10,8,4 ) <--> ( 6,9,1,7,3 )

Z  =  8707
Iteration count  = 146
-----------------------------------------------------

Final iteration count  = 180

Calculation time  =   00:00:00.77
```