

**NASA
Technical
Memorandum**

NASA TM - 103517

**AN IMPROVED EXPLORATORY SEARCH TECHNIQUE
FOR PURE INTEGER LINEAR PROGRAMMING
PROBLEMS**

By F.R. Fogle

Systems Analysis and Integration Laboratory
Science and Engineering Directorate

October 1990

(NASA-TM-103517) AN IMPROVED EXPLORATORY
SEARCH TECHNIQUE FOR PURE INTEGER LINEAR
PROGRAMMING PROBLEMS (NASA) 130 p CSCL 09B

N91-13910

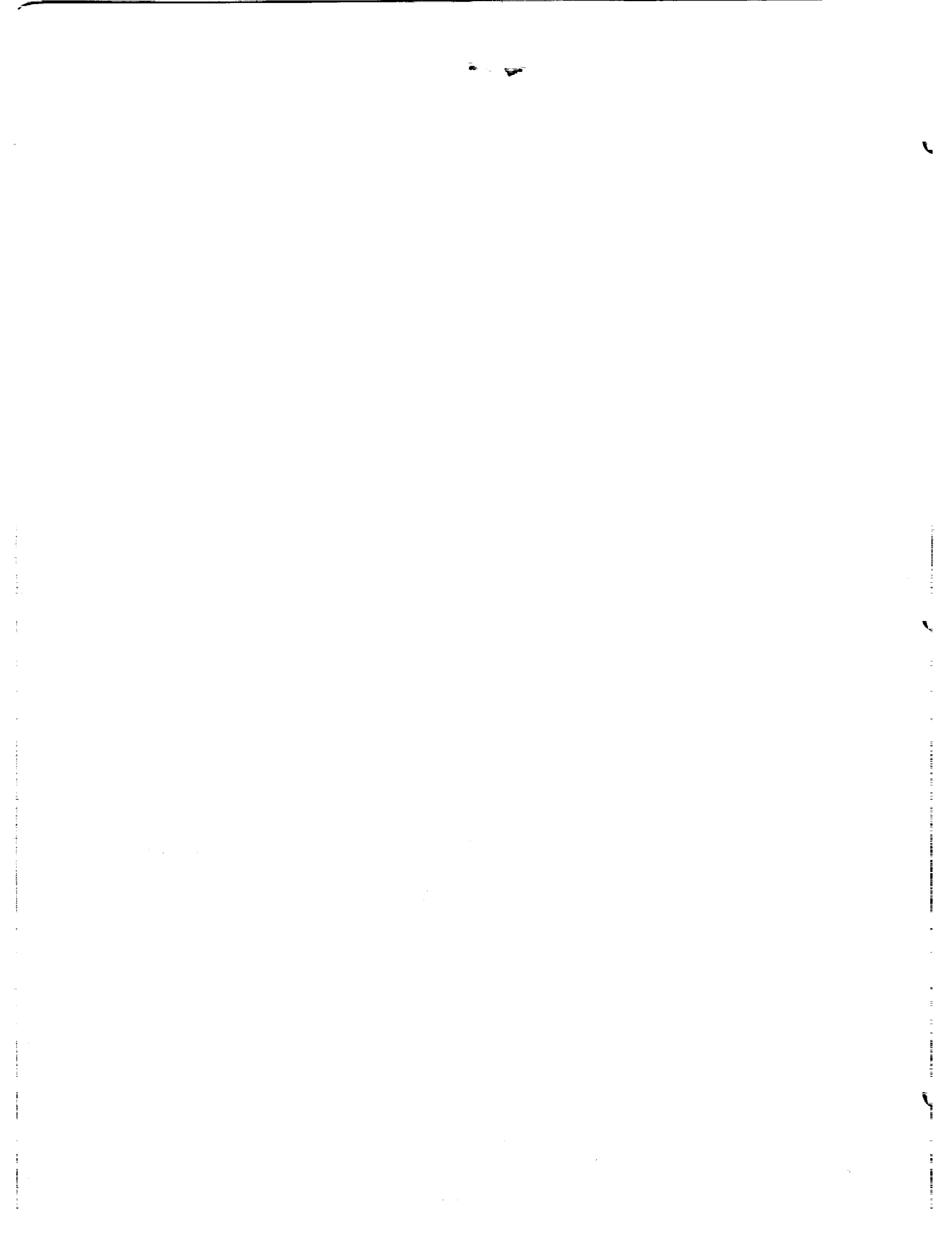
Unclass

G3/61 0319616



National Aeronautics and
Space Administration

George C. Marshall Space Flight Center





Report Documentation Page

1. Report No. NASA TM-103517		2. Government Accession No.		3. Recipient's Catalog No.	
4. Title and Subtitle An Improved Exploratory Search Technique for Pure Integer Linear Programming Problems				5. Report Date October 1990	
				6. Performing Organization Code	
7. Author(s) F.R. Fogle				8. Performing Organization Report No.	
				10. Work Unit No.	
9. Performing Organization Name and Address George C. Marshall Space Flight Center Marshall Space Flight Center, Alabama 35812				11. Contract or Grant No.	
				13. Type of Report and Period Covered Technical Memorandum	
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546				14. Sponsoring Agency Code	
				15. Supplementary Notes Prepared by Systems Analysis and Integration Laboratory, Science and Engineering Directorate.	
16. Abstract <p>This report documents the development of a heuristic procedure for the solution of pure integer linear programming problems. The procedure draws its methodology from the ideas of the Hooke and Jeeves type I and type II exploratory searches, greedy procedures, and neighborhood searches. It utilizes an efficient rounding procedure to obtain its first feasible integer point from the optimal continuous solution obtained via the simplex method.</p> <p>Since this procedure is based entirely on simple addition or subtraction of one to each variable of a point in n-space and the subsequent comparison of candidate solutions to a given set of constraints, it facilitates significant complexity improvements over existing techniques. It also obtains the same optimal solution found by the branch-and-bound technique in 44 out of 45 small to moderate size test problems. Two example problems are worked in detail to show the inner workings of the procedure. Furthermore, using an established weighted scheme for comparing computational effort involved in an algorithm, a comparison of this algorithm is made to the more established and rigorous branch-and-bound method. A computer implementation of the procedure, in PC-compatible Pascal, is also presented and discussed. This procedure for finding optimal solutions to integer-type problems may be applied to various systems engineering situations in the conceptual, preliminary, and detail design phases of the system development cycle.</p>					
17. Key Words (Suggested by Author(s)) Systems Engineering Integer Programming Optimization Techniques			18. Distribution Statement HEURISTIC METHODS OPERATIONS RESEARCH Unclassified - Unlimited		
19. Security Classif. (of this report) Unclassified		20. Security Classif. (of this page) Unclassified		21. No. of pages 131	22. Price NTIS

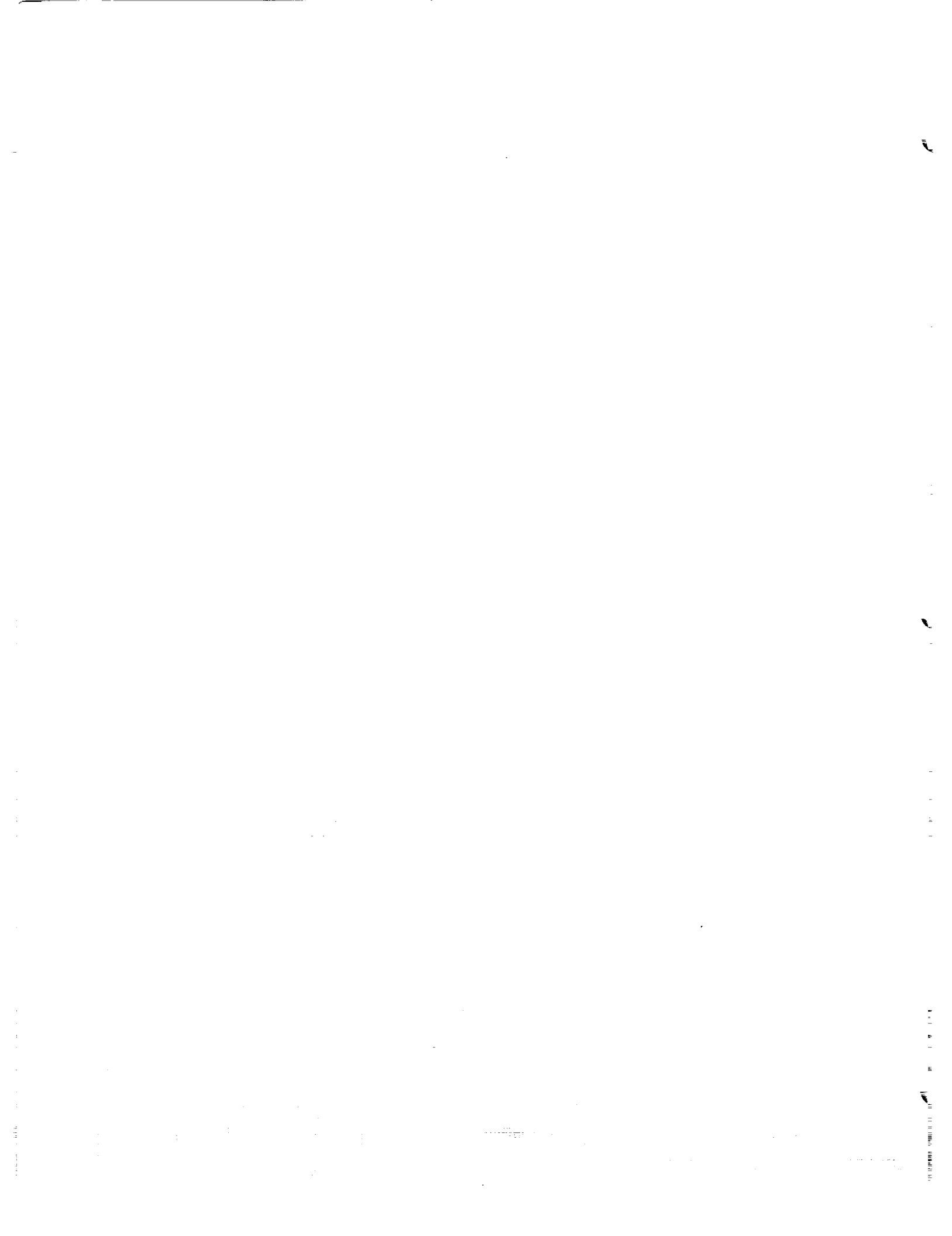


TABLE OF CONTENTS

	Page
I. INTRODUCTION	1
A. Background	1
B. Research Topic	2
C. Description of Succeeding Sections	2
II. LITERATURE REVIEW	3
A. Introduction	3
B. Integer Programming	3
1. General	3
2. Cutting Methods	5
3. Enumerative Methods	8
4. Heuristic, Approximate, and Direct Search Approaches	12
5. Nonlinear Problems	13
6. Rounding Procedures	14
7. Applications	16
C. Hooke and Jeeves Direct Search	16
1. Theory	16
2. Hooke and Jeeves Example	18
3. Improvements, Advantages, and Disadvantages	21
III. PROPOSED ALGORITHM – AN IMPROVED EXPLORATORY SEARCH I TECHNIQUE FOR INTEGER PROGRAMMING (IESIP)	22
A. Approach	22
1. General	22
2. Solution Procedure and Modifications Required to Existing Schemes	22
3. Advantages in Using the Proposed Procedure	28
B. Other Details of Algorithm	29
1. Rounding to Discrete Starting Solution	29
2. Constraint Involvement	30
3. Stopping Rule	30
C. Example Problems	32
D. Justification of Optimality	40
E. Computer Implementation	41
IV. COMPARISON OF RESULTS – COMPUTATIONAL EXPERIMENTS	44
A. Algorithm Performance Measures	44
1. Current Methods	44
2. Proposed Alternative	45

TABLE OF CONTENTS (Continued)

	Page
B. Algorithm Computational Complexity Scoring	46
1. Number of Operations	47
2. Weight Associated With Each Operation	49
C. Example Problem	50
D. Computational Experience With Test Problems	55
E. Multiple Optima	56
F. Restrictions Required of IESIP	58
V. CONCLUSIONS AND RECOMMENDATIONS	59
A. Conclusion	59
B. Recommendations for Future Work	60
REFERENCES	61
BIBLIOGRAPHY	65
APPENDIX A – Fractional Cut and Branch-and-Bound Examples	71
APPENDIX B – IESIP Pascal Computer Code	83
APPENDIX C – Test Problems	111

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Area of feasible integer solutions	6
2.	Gomory's cuts	7
3.	Cutting plane procedure block diagram	8
4.	Branch-and-bound tree diagram	10
5.	The branch-and-bound procedure	11
6.	Information flow diagram for direct minimization using Hooke and Jeeves pattern search	17
7.	Hooke and Jeeves exploratory and pattern moves	18
8.	Hooke and Jeeves exploratory search progression	23
9.	Modified type I exploratory search	25
10.	IESIP description flow diagram	27
11.	Two variable example problem	36

LIST OF TABLES

Table	Title	Page
1.	Sample input for IESIP computer program	42
2.	Sample output for IESIP computer program	43
3.	Worst case microprocessor instruction times	50
4.	Computational results of test problems	53
5.	Summary of test problem results	54

TECHNICAL MEMORANDUM

AN IMPROVED EXPLORATORY SEARCH TECHNIQUE FOR PURE INTEGER LINEAR PROGRAMMING PROBLEMS

I. INTRODUCTION

A. Background

Optimization, as related to mathematical programming, means to maximize (or minimize) an objective function of one or more variables subject to a set of confining functions or constraints. The distinguishing feature of discrete optimization, also referred to as integer programming or integer optimization, is that some or all of the variables are required to be in a discrete set, typically a subset of integers. These discrete (or integer) restrictions allow the mathematical representation of phenomena or alternatives where indivisibility is required or where there is not a continuum of alternatives.

The normal linear programming problem generally results in an optimal solution that has fractional (continuous) values for some or all of its decision variables. However, these fractional values in an optimal solution are not always acceptable as a realistic solution. For example, in a production optimization problem, 25.7 houses, 18.6 airplanes, or 5.4 machines may constitute optimal solutions, but do not represent useful results. Rounding off these results to produce integer solutions may lead to an infeasible solution, or such a solution may not be the global optimal integer solution for the problem. In these situations, only the integer programming problem and its solution can provide acceptable results. An integer programming problem where all the variables are restricted to integer values is called a "pure integer programming problem." A "mixed integer programming problem" has some variables that are restricted to integers while others are allowed to be fractional (continuous). There is also a special type of integer programming problem called the "zero-one integer problem," in which some or all decision variables are restricted to values of either zero or one.

Practical discrete (integer) optimization problems are very prevalent. An important and widespread area of application concerns the management and efficient use of scarce resources to increase productivity. These include operational problems such as the distribution of goods, machine scheduling, and production scheduling. They also include planning problems such as capital budgeting, facility location, and portfolio selection. Also included are design problems such as telecommunications, power system, transportation network design, VLSI circuit design, and the design of automated production systems. Integer optimization problems also arise in statistics (data analysis), physics (determination of minimum energy states), cryptography (designing unbreakable codes), and mathematics (proving combinatorial theorems). Moreover, applications of integer optimization are in a period of rapid development because of the widespread use of microcomputers and the data provided by information systems [1]. This rapid development is more pronounced in the manufacturing sector of the economy where increased competition and flexibility provided by new technology in the global marketplace make it imperative to seek better solutions from larger, more constraining, and more complex sets of alternatives.

Integer programming is one approach to problems of combinatorial optimization. A combinatorial optimization problem is defined as assigning discrete numerical values to a finite set of variables so as to maximize some objective function while satisfying a given set of constraints on the values the variables can assume. The transportation problem and the assignment problem are familiar examples. The transportation problem is an integer programming problem and the assignment problem is a zero-one programming problem. Elementary considerations can teach some important lessons regarding the numerical complexity and vastness these integer programming problems can possess. Any solution method that requires the enumeration of all 2^n possible combinations of "yes" and "no" decisions on n items may work well for up to 10 or 20 items. However, complete enumeration of all 2^n combinations requires 1,000 times as much work when n increases from 20 to 30 [2]. It is this potential cost of both time and money regarding enumeration of possibilities that has encouraged extensive theoretical research over the past three decades. This discussion will build on some of this theoretical research and also attempt to develop a new methodology. This new procedure will draw principally on the developments from the nonlinear programming arena, specifically the 1961 work of Robert Hooke and T. A. Jeeves.

B. Research Topic

The research topic documented in this report will establish a new methodology for solving pure integer programming problems by utilizing a modified version of the univariate exploratory move developed by Robert Hooke and T. A. Jeeves. Their technique was originally developed for application to nonlinear continuous problems. The method will also take some of its technique from the greedy procedure and the idea of unit neighborhoods. A rounding scheme will be determined which best suits the needs of these type problems. This scheme will use the continuous solution found by traditional methods (simplex or other suitable technique) and will create a feasible integer starting point. The Hooke and Jeeves exploratory search will be modified to accommodate integers and constraints and will then be employed to determine an optimal integer solution from the feasible starting solution. A user-friendly computer routine is presented that allows for rapid solution of problems up to 20 variables in size.

Two of the 45 test problems will be presented in detail. The remaining 43 problems are solved using the traditional branch-and-bound method and then compared by a computational complexity factor to the new technique. A separate set of appendices will document the software used, the test problems, and the details of two more accepted procedures for solving integer programming problems, fractional-cut, and branch-and-bound.

C. Description of Succeeding Chapters

The pertinent literature which was reviewed during this research is discussed in section II. The literature review was directed toward traditional integer programming techniques and the theory behind the exploratory and pattern searches developed by Robert Hooke and T. A. Jeeves. Also discussed are any relevant data on research regarding the use of direct search techniques on integer programming problems.

Section III discusses in detail the new algorithm, Improved Exploratory Search Integer Programming (IESIP). Examples of small variable and constraint size are presented to show the technique's inner workings. Also shown is the justification for accepting the solution found as the optimal.

Section III also provides the computer implementation of the technique. Section IV extends this discussion to examples and compares the new technique on a basis of computational complexity to the branch-and-bound method. Some of the same examples presented in section III are revisited along with larger and more difficult problems.

In section V, conclusions are drawn from the research and computational experiments. Recommendations and suggestions for future research are also presented.

II. LITERATURE REVIEW

A. Introduction

The focus of the literature review was on the following topical areas: traditional integer programming techniques, heuristic and approximate techniques, integer rounding, nonlinear techniques, and the theory and methodology behind the Hooke and Jeeves direct search technique.

A review of the 1960 through September 1989 "Dissertations Abstracts International" (Science and Engineering) substantiated the belief that no dissertations have been written that specifically present the methodology shown in this report for the solution of integer programming problems. The abstracts were examined back to 1960, since this was the general timeframe of the inception of both integer programming and the Hooke and Jeeves procedure. Further examination of technical literature during the same time period was conducted by utilizing an online computer search available at the Redstone Scientific Information Center (RSIC). It also revealed no articles on the proposed subject. The only related areas of discussion were direct search techniques (which will be examined in more detail in section B.4), but none of them used the proposed procedure as their methodology.

B. Integer Programming

1. General

As stated earlier, integer programming concerns itself with the class of optimization problems in which some or all the variables are required to be integer. More specifically, it can be stated that any decision problem with an objective function to be maximized (or minimized) in which the variables must assume nonfractional or discrete values may be classified as an integer optimization problem [3].

Integer programming confines itself to a specific group of problems that are part of a larger classification known as "mathematical programming" problems. Garfinkel and Nemhauser [4] present the following notation in representing mathematical and integer programming problems:

A general mathematical programming problem can be stated in the form

$$\max f(x) ; \quad x \in S \subseteq R^n \quad (1)$$

where R^n is the set of all n -dimensional vectors of real numbers and f is a real-valued function defined on S , where S is called the constraint set and f is called the objective function. Every $x \in S$ is a feasible solution to equation (1). An integer programming problem is a mathematical programming problem in which

$$S \subseteq Z^n \subseteq R^n$$

where Z^n is the set of all n -dimensional integer vectors.

For the linear case, an integer programming problem is formulated as

$$f(x) = cx$$

and

$$S = \{x | Ax = b, x \geq 0 \text{ integer}\} \quad (2)$$

where A is an $m \times n$ matrix; b is an m -vector; c is an n -vector; and 0 is an n -vector of zeros. The set S is a convex set of linear constraints which is also mathematically referred to as a polyhedron or polytope. The more standard form of the integer linear programming problem (ILP) is

$$\begin{aligned} & \max (\min) cx \\ & \text{subject to } Ax = b \\ & x \geq 0 \text{ and integer,} \end{aligned} \quad (3)$$

or written in summation notation,

$$\max (\min) \sum_{j=1}^n c_j x_j \quad (4)$$

$$\text{subject to } \sum_{j=1}^n a_{ij} x_j \begin{pmatrix} \leq \\ = \\ \geq \end{pmatrix} b_i, \quad i = 1, 2, \dots, m$$

$x_j \geq 0$ and integer, $j = 1, 2, \dots, n$.

This is the standard linear formulation of integer programming problems. Nonlinear problems and their current solution methods will be discussed later. If all variables x_j in equation (4) are restricted to integer values, the problem is referred to as a "pure" integer problem. Otherwise, if some x_j values can take on real values, then the problem becomes a "mixed" integer problem.

Much progress has been made in developing procedures for handling these type problems. Since the pioneering work of Ralph Gomory in the late 1950's, integer programming has been an exciting and rapidly developing area of operations research [4]. The past three decades have witnessed extensive theoretical research in this area. The result is a vast collection of solution methods and algorithms and numerous applications to real-world problems.

Current methods for handling integer programming problems generally are categorized into the following two broad types: (1) cutting methods and (2) search (or enumerative) methods. Cutting methods systematically add special "secondary" constraints to the continuous optimum, which represent necessary conditions for integrality. The continuous solution space is modified by these cuts until its continuous optimum extreme point satisfies the integer conditions. This group of IP methods gets its name from the fact that the added "secondary" constraints cut (or eliminate) parts of the solution space that do not contain feasible integer points. In contrast, search methods have their roots in the idea of enumerating all feasible integer points. The basic idea is to test only a small portion of the feasible integers explicitly, but also account for the remainder implicitly. The now famous branch-and-bound technique, originally developed by A. H. Land and A. G. Doig in 1960, is the most common of the search methods. It, like the cutting methods, starts with a continuous optimal solution, but then systematically "partitions" the solution space into smaller subproblems by deleting parts that are integrally infeasible [5].

The above-mentioned categories of integer programming problems, cutting methods, and search methods, will now be discussed separately and in detail.

2. Cutting Methods

The optimal solution to an integer linear programming problem can very well be an interior point of the set of all feasible solutions. Therefore, algorithms (e.g., simplex) that explore only the extreme points of this set, will be unable to locate such a solution. One way to eliminate this problem is by cutting off sections of the initial set of feasible solutions with hyperplanes such that the optimal integer solution becomes an extreme point of the new set of feasible solutions. Hyperplanes so used are called "cutting" planes. There are many ways and variations on methods for obtaining these cutting planes, but this discussion will concentrate mostly on the original method developed in 1958 by Ralph Gomory [6].

Cutting planes were used as early as 1952, when Dantzig, Fulkerson, and Johnson began work on the familiar traveling salesman problem. They employed subtour elimination constraints as well as ad hoc linear constraints generated to exclude a current fractional answer. By 1958, Gomory had developed his finitely convergent cutting plane method [7]. In 1960, Gomory produced a second cutting plane algorithm for integer programming that computationally requires only additions and subtractions [8].

As stated earlier, the cutting plane method attempts to pare down the feasible solution space so that it contains integer corner points. Figure 1 is a graphic representation of a typical two-variable feasible solution area for a maximization problem.

The cutting planes reduce the area of feasible solutions to “force” integer corner points. Three cutting planes (1, 2, and 3) have been added in figure 2, which slice into the solution space, thereby eliminating some of the shaded portion in the original solution space. The boundaries of the reduced area of feasible solutions are said to make up the “convex hull.” This is defined as the smallest convex set necessary to include all the feasible integer points [5].

The detailed mechanics of how cutting plane equations are developed are illustrated by example in appendix A. Here we will only offer a feeling for how the procedure works. First, cutting planes are developed in order to force the noninteger valued variables to integer values. If a starting solution involves a number of noninteger variables, then usually one variable is selected to provide the basis for developing a new constraint (cutting plane). Once the constraint has been

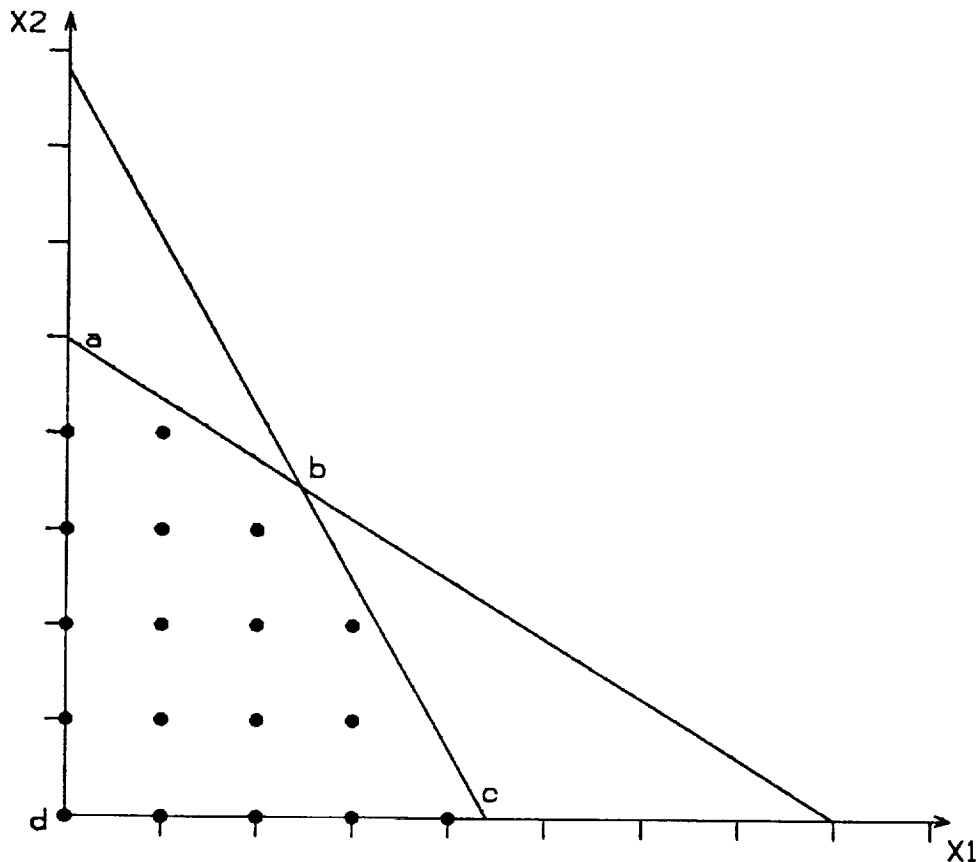


Figure 1. Area of feasible integer solutions.

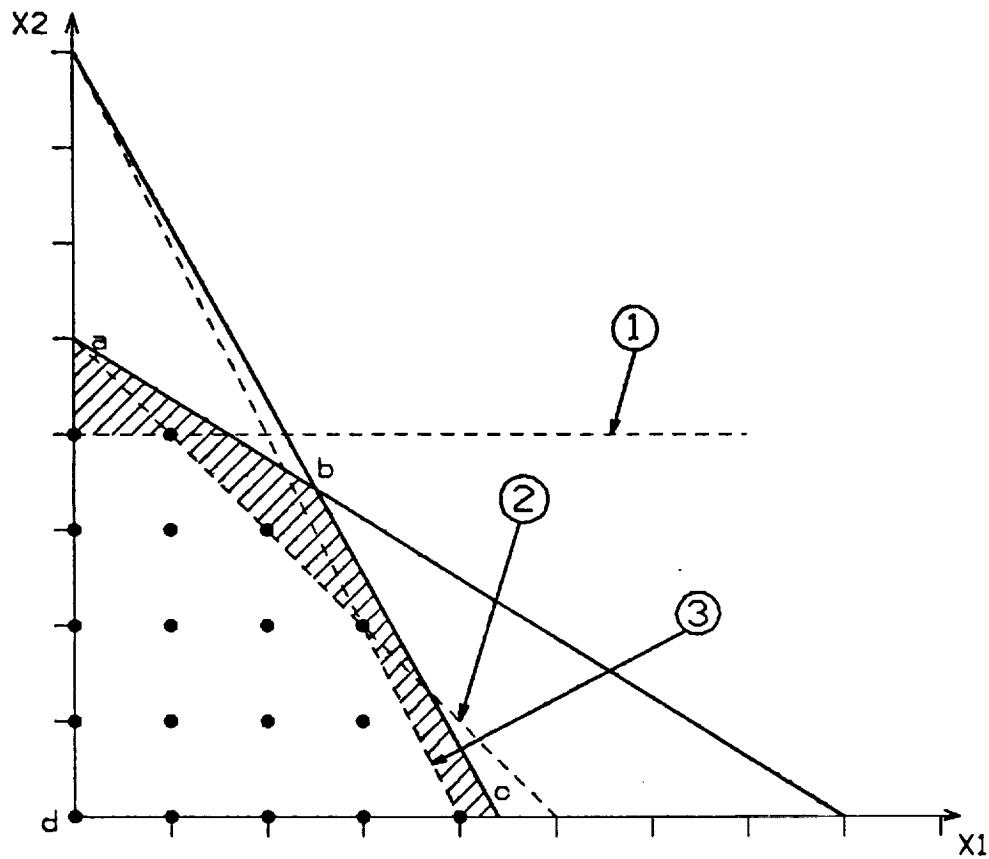


Figure 2. Gomory's cuts.

identified, it can be added to the original constraint cut and then, by utilizing the dual simplex method, the constraint is incorporated in the final tableau of the previous solution. This allows the optimal solution of the larger problem to be found without starting from scratch. Sometimes, it is not necessary to add enough cutting planes to reduce the solution space to its convex hull. Depending on the objective function, it may take as little as one cutting plane to reach the optimal integer solution. This algorithm is sometimes referred to as the "fractional method" because all the nonzero coefficients of the generated cut are less than one. The block diagram in figure 3 illustrates the Gomory cutting plane procedure in a logical flow process. The fractional cutting method is applicable to pure integer programming problems while its counterpart, "the mixed algorithm," is designed for the mixed integer problem.

Many modifications and extensions to Gomory's basic algorithm have been developed since 1960. In 1965, Glover produced a dual algorithm for solving pure integer problems. The general idea of Glover's method is to find the optimal integer solution by determining lower bounds on each variable in such a way as to satisfy a necessary integrality condition. Ben-Israel and Charnes were the first to suggest a primal cutting algorithm along the same ideas of Gomory's all integer cutting method in 1962. The basic difference is that primal (integer) feasibility is maintained at all stages of calculations. Glover and Young developed the primal cutting plane algorithms further in 1968 and 1971, respectively [3].

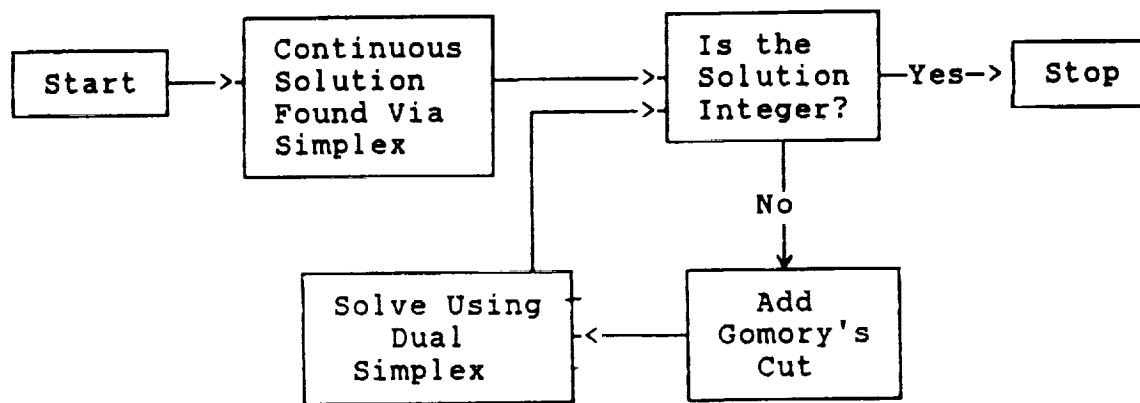


Figure 3. Cutting plane procedure block diagram.

No single cutting procedure can be considered uniformly superior from a computational standpoint, although in isolated cases of specially structured problems cuts have proven effective. The general feeling among most experts is that cutting methods should not be relied upon to solve integer problems regardless of size. Experience has shown that some rather small problems could not be solved by cutting methods. For instance, cases have been reported of random changes in constraint order causing computationally easy problems to become extremely formidable [5]. Certain problems, however, tend to be solved easily using cuts such as Gomory's original fractional cutting plane method. Among these are set covering problems such as those encountered in a typical crew scheduling problem.

3. Enumerative Methods

There is no generally accepted terminology for the class of integer programming methods known as branch-and-bound, search, or enumerative procedures. Each of these titles refers to methods for solving integer programming problems by breaking up the feasible set into subsets, calculating bounds on the objective function value over each subset, and using the bounds to discard certain subsets of solutions from consideration. However, within this general class of methods, one can distinguish two basic prototypes. One, introduced by Land and Doig [9] in 1960, and later modified by Dakin in 1965 [10], and Driebeck in 1966 [11], is aimed at solving several pure and mixed integer problems and uses linear programming as its main vehicle. The other one, typified by Balas in 1965 [12], Lemke and Spielberg in 1967 [13], and Geoffrion in 1969 [14], is concerned only with solving 0-1 problems and uses as its main tool logical tests exploring the implications of the binary nature of the variables or inequalities. Most practitioners prefer to reserve the term branch-and-bound for the first of the above two approaches while calling the second one "implicit" enumeration. Others consider this distinction less and less relevant with the passage of time, as the two approaches are increasingly borrowing from each other, to the extent that the more recent algorithms usually contain elements of both.

The branch-and-bound techniques like the cutting plane methods solve integer programming problems by first considering their continuous solution. But, as mentioned earlier and in contrast to cutting methods, branch-and-bound applies itself directly to both "pure" and "mixed" problems. The general procedure for the branch-and-bound method will briefly be discussed below and a detailed example problem presented in appendix A.

The branch-and-bound method for the solution of a constrained integer problem uses basically the following steps:

- **Branching Step:** The solution starts with the partitioning of all feasible solutions into smaller subsets, each representing a subproblem of the original problem.
- **Bounding Step:** Then the method finds, for a given subset, a lower bound value Z for an objective function (maximization problem). Usually, this is the value of the objective function for the best feasible integer solution found so far.
- **Fathoming Step:** After each branching and bounding step, the method excludes a particular subset from further consideration if (1) the subset has no feasible solution, (2) the subset is feasible but has a lower bound value less than or equal to the lower bound value of a feasible solution known to date, and (3) the subset has already reached its best feasible solution.
- **Terminating Step:** The partitioning of all feasible solutions of the original problem continues through the repetition of the first three steps. When a feasible solution is found for any subset that is higher than the best lower bound value known to date, the new value becomes the new lower bound of the original problem. At the end of this systematic search process, the lower bound value, which remains uncontested, determines the optimal solution of the problem.

The history and systematic approach to branching-and-bounding of the subsets can be shown in a tree diagram similar to the example problem shown below and in figure 4. Each node of this diagram represents a subset defined by a subproblem.

$$\text{maximize } Z = f(x_1, x_2) = 2x_1 + 8x_2$$

$$\text{subject to: } 2x_1 - 6x_2 \leq 3$$

$$-1x_1 + 4x_2 \leq 5$$

$$2x_1 + 2x_2 \leq 13$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

The flow chart depicted in figure 5 summarizes the branch-and-bound procedure.

Branch-and-bound methods have historically been more readily applied to problems for the following two reasons: (1) they can be developed and modified to enumerate only a portion of all candidate solutions while automatically discarding the remaining points as nonpromising, and (2) they lend themselves readily to computer-based solutions [16]. The one major disadvantage of this

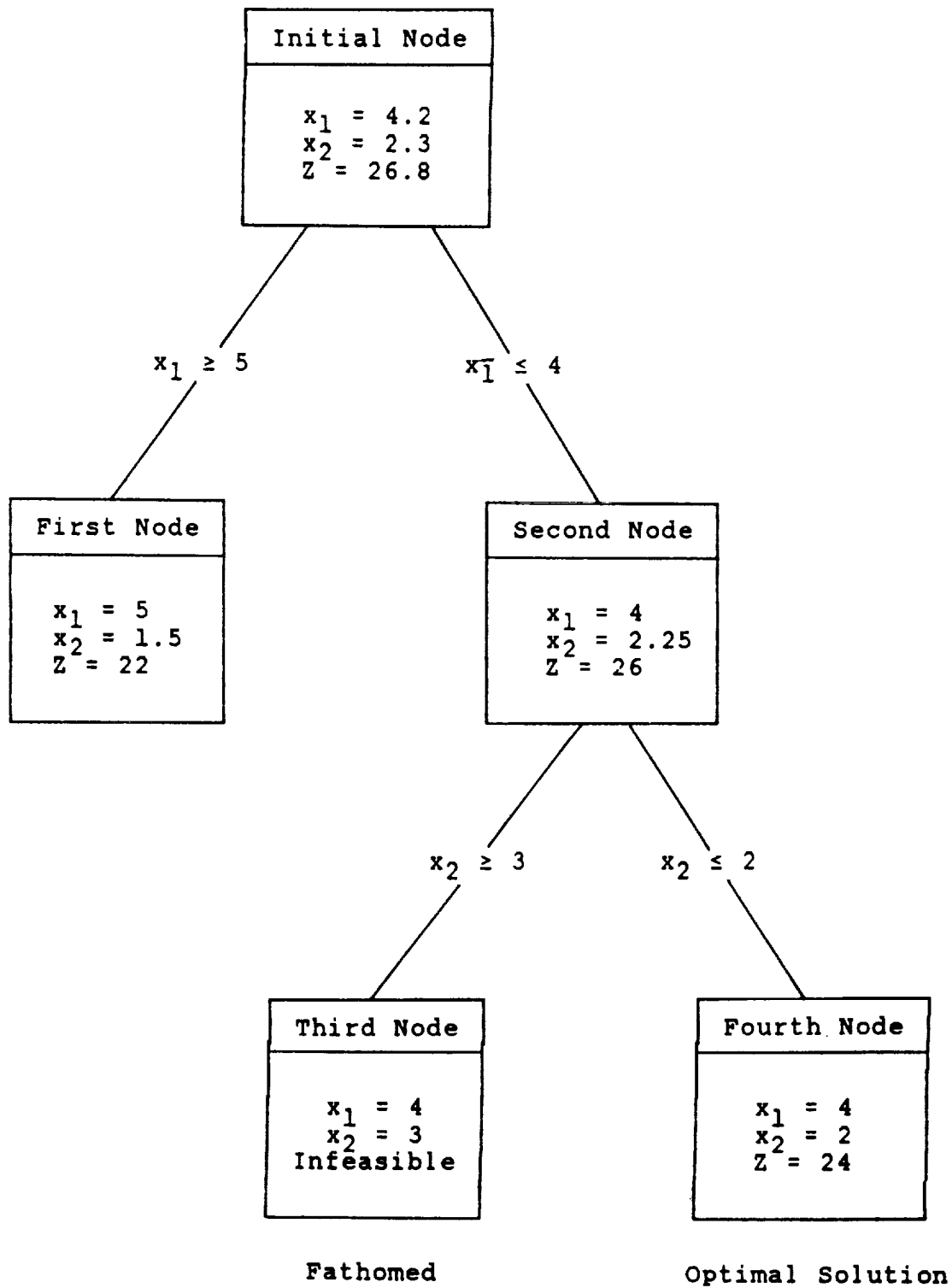


Figure 4. Branch-and-bound tree diagram.

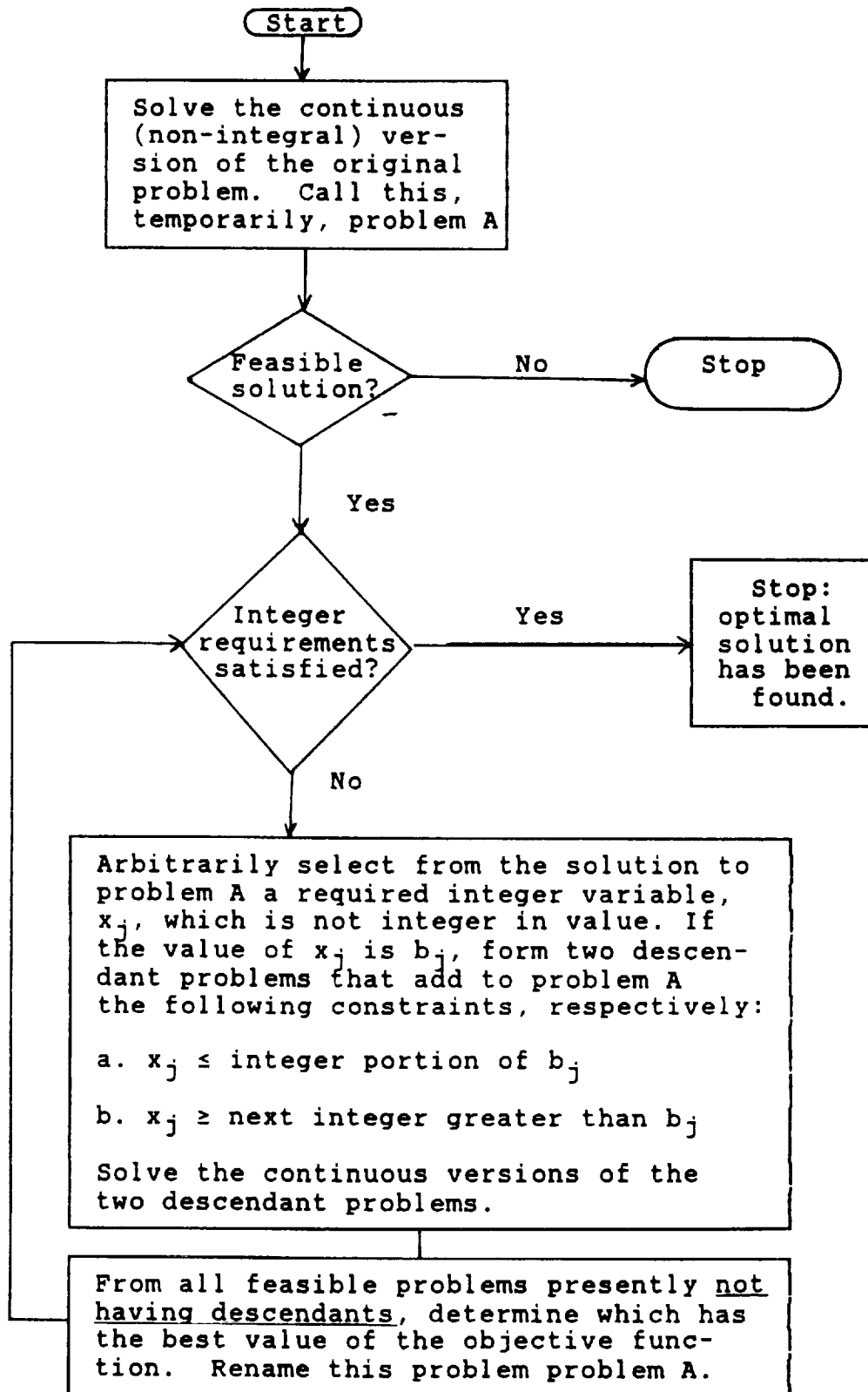


Figure 5. The branch-and-bound procedure [15].

class of methods is that it is necessary to solve a complete linear programming problem at each node. In spite of this, branch-and-bound methods are the most effective in solving integer programming problems of practical size. Most all available commercial codes are based on these methods. This does not mean that all integer-type problems are solved with branch-and-bound methods, but rather that if given a choice between a cutting plane method and a branch-and-bound, the latter is usually utilized [5].

4. Heuristic, Approximate, and Direct Search Approaches

Another approach to solving integer programming problems that has received attention has been the use of nonexact (heuristic or approximate) procedures. Many of these techniques give an approximate solution to the problems. They are designed to provide a "good" solution, but in most cases cannot guarantee optimality.

Methods of this kind are valuable for several reasons. First, computational experience with (exact) algorithms has sometimes not been the best on certain types of problems. Many real-world problems of interest are too large to be solved exactly. Secondly, enumerative algorithms invariably benefit from beginning with a good feasible solution. Finally, a feasible solution can provide a lower bound on the optimal objective function value. This bound can be used for fathoming in enumerative algorithms and in cutting plane algorithms as a cut or as a source row for a cut [4].

Search techniques for discrete optimization include those developed by Healy in 1964 [17], Reiter and Sherman in 1965 [18], Reiter and Rice in 1966 [19], Kreuzberger in 1970 [20], and Kochenberger, McCarl, and Wyman in 1974 [21].

The Reiter and Sherman approach combines an intelligent search with a random search. It starts with any point and then by using a suitably chosen local search technique, it moves to better points until eventually no further improvement is possible. A set of local optima is generated. The local optima are then sampled sequentially according to a plan (suitable probability distribution) that stops when the expected return from further sampling is not sufficient to pay the cost of further sampling. Reiter and Rice proposed a solution procedure consisting of (a) choosing a random starting point, (b) locating a feasible point, and (c) applying a modified gradient maximizing procedure.

Developments in direct search methods and developments in cutting methods and enumeration techniques reached their high point in volume in the early 1970's. Recently though, progress has been refocused on efficient approximation and direct search heuristic procedures for finding feasible integer solutions for integer programming problems. These solutions are not necessarily optimal, but usually will be better than can be found by simple rounding [22].

A new approach has been the combining (or hybridization) of two proven mathematical programming techniques to solve integer programming problems. These two methods are dynamic programming and some proven integer method such as branch-and-bound. The dynamic programming methodology is used to search candidate hyperplanes efficiently for the optimal feasible integer solution. Relaxations and fathoming criteria, which are fundamental to branch-and-bound, are incorporated within the separation and initial fathoming provided by the dynamic programming

framework. This idea was explained and tested in the late 1970's by Cooper and Cooper [23] and also by Marsten and Marin [24]. Their computational results for small to moderate size problems showed promising results for the hybrid technique advocates.

Important to the development of the procedure proposed here are two heuristic approaches known as greedy procedures and local improvement schemes. In a general sense, a "greedy" procedure is one in which the decision maker selects at each stage of the process an alternative that is best among the feasible alternatives without regard to the impact the choice may have on subsequent decisions. The word "best" implies the most favorable with respect to the objective function. Strict greedy procedures have been applied to traveling salesman problems, minimal spanning tree problems, and knapsack problems. The greedy algorithm does what is locally best without regard to future consequence. Therefore, for most integer optimization problems, greedy algorithms as they were designed are merely heuristics for finding a good feasible solution [1].

Local improvement schemes form the core of most continuous optimization procedures. The search proceeds by sequential improvement of problem solutions, advancing at each step from a current solution to an objective function superior neighbor. These are generally called local improvement searches in the discrete optimization arena. These are also referred to as local optimization or neighborhood searches. The concept of the neighborhood of a point x^* in Euclidean space is defined to be an open hypersphere containing x^* . Because the variables are required to be integer, different kinds of neighborhoods are needed. In general, every neighborhood of an integer n -vector x^* will be a set of integer vectors including x^* , and in some sense near x^* . The neighborhood defined below is the most applicable to this discussion [4]. The unit neighborhood of x^* is defined as

$$R(x^*) = \{x \mid x_j = x_j^* - 1, x_j^*, x_j^* + 1\} \text{ for } j = 1, 2, \dots, n .$$

Local improvement schemes can sometimes be used as a second phase in an optimization procedure that begins with the output of the greedy procedure [25].

Garfinkel and Nemhauser [4] suggest a local optima scheme that feeds from both the greedy procedure and the local improvement or neighborhood search idea. However, just as stated above, at each iteration only the best alternative is chosen and examined again. This feature and sometimes pitfall of these procedures is discussed and improved upon in the proposed algorithm presented in section III.

5. Nonlinear Problems

In view of the relative computational difficulties of algorithms for the solution of integer linear programming problems, it is not surprising that the situation is, in general, even worse for nonlinear integer programming (NLIP). While algorithms do exist for the solution of certain classes of nonlinear integer programming problems, most have not been tested computationally and those that have are effective only for relatively small-size problems.

Nonlinear problems that have separable objective functions and linear constraints have been approached in many instances by dynamic programming techniques or implicit enumeration. The value of a dynamic programming approach is that it yields a global optimum. The difficulty associated with dynamic programming approaches is that multiple constrained problems lead to multiple dimensional tables for the dynamic programming return functions, and this, by its nature, leads to unwieldy storage and computational requirements [26]. Pegden and Petersen [27] present a generalized implicit enumeration method for solving separable NLIP problems with linear constraints and report computational results. Their method uses a combination of search techniques and linear programming approximation to gradually tighten bounds on the variables. The ordering of the variables is an important part of their method. Cabot and Erenque [28] also present a branch-and-bound method for solving the problem of minimizing a separable concave function over a convex polyhedral set where the variables must be integer valued.

Another type of nonlinear integer problem that has received much attention has been a problem with a quadratic objective function. It was first addressed by Balas [29] in 1969. He presents an algorithm based on duality theory for both pure integer and the mixed integer case. Balas also extends the algorithm further to include integer nonlinear problems with convex objective function and constraints. Other solution methods for quadratic integer programming methods were developed later by McBride and Yormark [30] in 1980, and by Volkovich, Roshchin, and Sergiando [31] in 1986.

Most literature on nonlinear methods pointed to dynamic programming and branch-and-bound methods, and to a lesser extent ideas from implicit enumeration and cutting plane techniques. In any method, the difficulty of dealing with nonlinear problems grows even more acute with an increase in problem size. Cooper [26] suggests that a new methodology is needed that would be completely different in a theoretical sense.

6. Rounding Procedures

On the surface it seems possible to treat a problem of integer programming as one of ordinary linear programming and to solve it by the standard simplex method. However, in order to obtain an integer solution, we must either truncate or round off the solution obtained by the simplex method. Sometimes this is adequate but, unfortunately, there are often pitfalls to this approach. One is that the optimal linear programming solution is not necessarily feasible after it is rounded. Another is the fact that there is no guarantee that this rounded solution will be the optimal integer solution even if it passes the feasibility criterion. To illustrate this, the following example is appropriate:

$$\text{maximize } Z = f(x_1, x_2) = x_1 + 5x_2$$

$$\text{subject to: } x_1 + 10x_2 \leq 20$$

$$x_1 \leq 2$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

The linear programming optimal solution is $x_1 = 2$, $x_2 = 9/5$, and $Z = 11$. If we round $9/5$ to 1, then the resulting integer solution is $x_1 = 2$, $x_2 = 1$, which yields $Z = 7$. This is far from the optimal integer solution of $Z = 10$ at $x_1 = 0$, $x_2 = 2$ [22].

Even though rounding has its pitfalls, it has still seen much attention for its applicability to obtain a starting feasible solution or as a method to obtain approximations to optimal solutions. Early work was performed by Wagner, Giglio, and Galser in this area by applying rounding heuristics to obtain an approximation to an optimal solution when dealing with preventive maintenance scheduling [32]. Giglio and Wagner also explored the idea of rounding to integer values to approximate solutions in machine scheduling problems [33]. In both cases, the computational results of many examples were statistically analyzed in reference to the approximate solution's closeness to the optimal solution. With these approaches, the trade-off between decreased computation time (decreased by up to 15 percent) versus an "approximation" to an optimal solution would have to be considered on a case-by-case basis. In certain applications, an approximate answer is not good enough regardless of how much computer time is saved.

In 1980, the idea of rounding reemerged with an attempt by John Bartholdi to round to integer values and obtain a solution to within a specified bound [34]. Bartholdi suggested that if x_{LP} is the optimal solution to the linear program, then the heuristically rounded solution is

$$x_h = [x_1], [x_1 + x_2] - [x_1], [x_1 + x_2 + x_3] - [x_1 + x_2], \dots, [\sum^n x_k] - [\sum^{n-1} x_k]$$

where x_1, x_2, x_3, \dots are the solution variables for the continuous solution, and $[x_i]$ is the smallest integer greater than or equal to x_i (the rounded-up value). Bartholdi's application of his method was limited to 0-1 problems in cyclic scheduling and set covering type situations.

The simplest approach to rounding was put forth by Baum and Trotter [35] recently and consists of rounding up in a minimization problem and rounding down in a maximization problem. They concede that under constraints this procedure will not always give a feasible solution. This was illustrated by the maximization example presented at the beginning of this discussion.

Taha sums up the limitations of utilizing rounding as follows:

1. If a feasible solution is obtained by rounding, one should not be under the illusion that such a solution is optimal or even close to optimal. The rounding procedure at best may be regarded as heuristic.

2. Any integer model having an original equality constraint can never yield a feasible integer solution through rounding. This is based on the assumption that only basic variables can be rounded, if necessary, and that all the nonbasic variables remain at zero level [3].

7. Applications

Since its inception in the 1950's, integer programming has become the tool used by many engineers and scientists to solve numerous real-world problems. As already mentioned, many situations yield programming formulations with some or all of the variables required to be integer. Included in these are scheduling, location, network, and selection problems, which appear in industry, military, education, health, and other environments. Specific examples include problems of facility location, resource-task scheduling, traveling salesman, capital budgeting, knapsack (cargo loading), and production-storage-distribution [16].

Specific applications of integer programming to every-day industrial problems are also far reaching. Integer programming techniques have been used to solve everything from electrical power system design/expansion [36–39] to the problem of determining measurements of a given number of sizes of apparel so as to maximize expected sales or minimize an index of aggregate discomfort [40]. Integer programming has been used extensively in the investment portfolio selection arena to optimize returns of stock option strategies [41,42]. It has also been used in determining a city's optimal bus crew scheduling [43] and a power company's optimal generator maintenance schedule [44].

C. Hooke and Jeeves Direct Search

1. Theory

Conceptually, the simplest type of search method is one that changes one variable at a time while keeping all the others constant until the minimum or maximum is reached. For example, one method would be to set one of the variables, say x_1 , constant and vary x_2 until a maximum or minimum was obtained. Then, keeping a new value of x_2 constant, change x_1 until an optimum for the value of x_1 is achieved, and so on. The direct search method of Robert Hooke and T. A. Jeeves is just such a univariate sequential technique of which each step consists of two kinds of moves. The technique alternates sequences of local univariate exploratory moves with extrapolations (or pattern moves, as Hooke and Jeeves refer to them) [45]. The basis for the method is the intuitive presumption that a strategy that was successful in the past will be successful in the future.

The development of the algorithm assumes the following:

$$\text{Minimize: } f(x) \quad x \in E^n$$

where $x = (x_1, x_2, \dots, x_i)$, E^n represents n -dimensional Euclidean space, and $f(x)$ can be either linear or nonlinear in nature. Its procedure is illustrated with a flow diagram in figure 6. The algorithm operates in the following manner:

Initial values for all the elements of x must be provided as well as an initial incremental change ϵ_i . To initiate an exploratory search, $f(x)$ is evaluated at a base point (the base point is the vector of initial guesses of the independent variables for the first cycle). Then each variable is

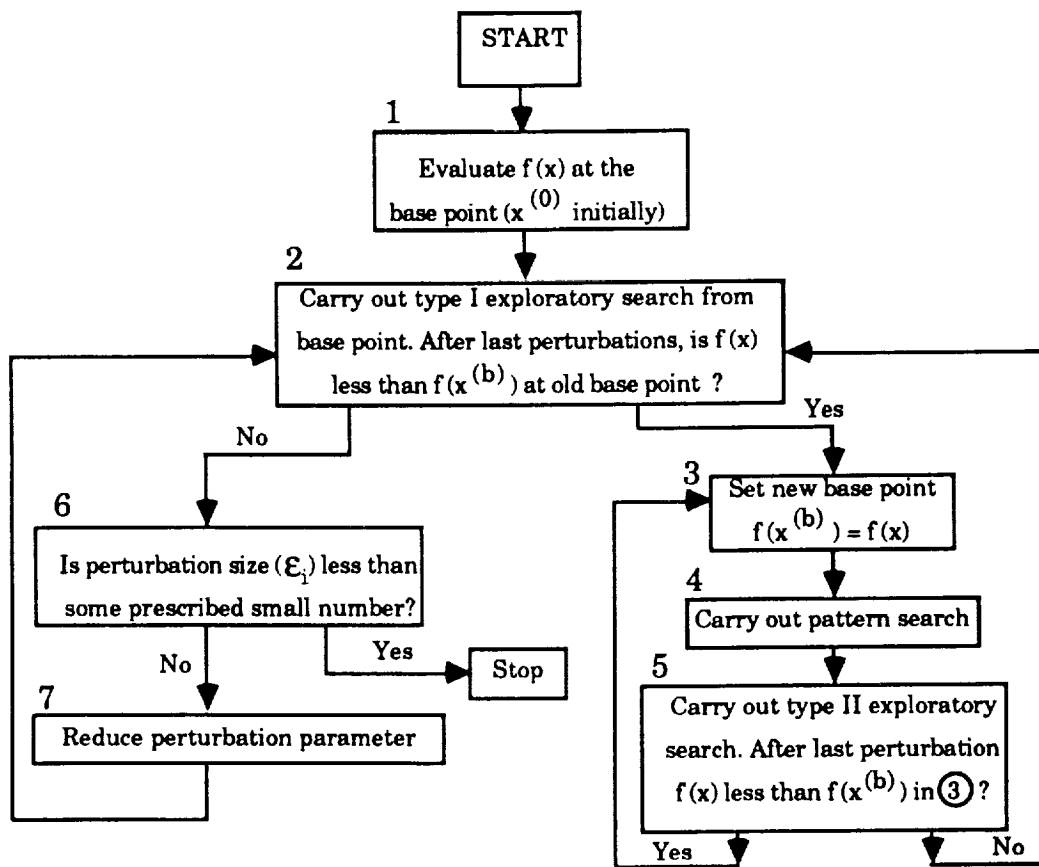


Figure 6. Information flow diagram for direct minimization using Hooke and Jeeves pattern search.

changed in rotation, one at a time, by incremental amounts, until all the parameters have been so changed. To be specific, $x_1^{(0)}$ is changed by an amount $+\epsilon_1$, so that $x_1^{(1)} = x_1^{(0)} + \epsilon_1$. If $f(x)$ is reduced (minimization), $x_1^{(0)} + \epsilon_1$ is adopted as the new element in x . If the increment fails to improve the objective function, $x_1^{(0)}$ is changed by $-\epsilon_1$, and the value of $f(x)$ is again checked as before. If the value of $f(x)$ is not improved by either $x_1^{(0)} \pm \epsilon_1$, $x_1^{(0)}$ is left unchanged. Then $x_2^{(0)}$ is changed by an amount ϵ_2 , and so forth, until all the independent variables have been changed to complete one exploratory search. For each step or move in the independent variable, the value of the objective function is compared with the value at the previous point. If the objective function is improved for the given step, then the new value of the objective function replaces the old one in the testing. However, if a perturbation is a failure, then the old value of $f(x)$ is retained.

After making one (or more) exploratory searches in this fashion, a "pattern search" is made. The successfully changed variables (i.e., those variable changes that decreased $f(x)$) define a vector in E^n (n -dimensional Euclidean space) that represents a successful direction for minimization. A series of accelerating steps, or pattern searches, is made along this vector as long as $f(x)$ is decreased by each pattern search. The magnitude of the step for the pattern search in each coordinate direction is roughly proportional to the number of successful steps previously encountered in each coordinate direction during the exploratory searches for several previous cycles. Therefore, the pattern moves increase in length as long as the search proceeds in the same direction. Furthermore, the exploratory sequences become increasingly farther apart as long as the search proceeds in the same direction. Exploratory moves and pattern moves are illustrated graphically in figure 7. An

Type I Exploratory Search.

<u>Variable</u>	ϵ_i	
$x_1 = 2.00 + 0.60 = 2.60$		$f(2.6, 2.8) = 0.048$ failure
$x_1 = 2.00 - 0.60 = 1.40$		$f(1.4, 2.8) = 0.073$ success
$x_2 = 2.80 + 0.84 = 3.64$		$f(1.4, 3.64) = 0.052$ failure
$x_2 = 2.80 - 0.84 = 1.96$		$f(1.4, 1.96) = 0.104$ success
$x^{(1)} = (1.4, 1.96)$		

Type I was a success. Therefore, new base point = (1.4, 1.96).

A \rightarrow Pattern Search is made from (1.4, 1.96) according to

$$x_i^{(k+1)} = 2x_i^{(k)} - x_i^{(b)}$$

where $x_i^{(b)}$ is the old base x vector, which is now $x^{(0)}$

$$x^{(2)} = 2(1.4, 1.96) - (2, 2.8)$$

$$x^{(2)} = (0.80, 1.12)$$

$$f(0.80, 1.12) = 0.22 \quad .$$

Now a type II exploratory search is made, with success or failure based on comparison to $f(0.8, 1.12) = 0.22$.

<u>Variable</u>	ϵ_i	
$x_1 = 0.80 + 0.60 = 1.40$		$f(1.40, 1.12) = 0.14$ failure
$x_1 = 0.80 - 0.60 = 0.20$		$f(0.20, 1.12) = 0.38$ success
$x_2 = 1.12 + 0.84 = 1.96$		$f(0.20, 1.96) = 0.19$ failure
$x_2 = 1.12 - 0.84 = 0.28$		$f(0.20, 0.28) = 0.67$ success
$x^{(3)} = (0.20, 0.28) \quad .$		

To determine if the pattern search A was a success, $f(0.20,0.28)$ is compared to $f(1.4,1.96)$, i.e., $0.67 < 0.104$. Therefore, the pattern search was a success and the new base point is

$$x^{(3)} = (0.20,0.28) \rightarrow f(0.20,0.28) = 0.67$$

and the old base point is now

$$x^{(1)} = (1.4,1.96) \rightarrow f(1.4,1.96) = 0.104 .$$

$B \rightarrow$ Since success has occurred, we proceed with another pattern search.

$$x^{(4)} = 2x^{(3)} - x^{(1)}$$

$$x^{(4)} = 2(0.20,0.28) - (1.4,1.96)$$

$$x^{(4)} = (-1, -1.4) \quad f(-1, -1.4) = 0.51 .$$

A type II exploratory search is now performed on this new point to determine if the pattern move was successful.

Variable

ϵ_i

$$x_1 = -1 + 0.6 = -0.40 \quad f(-0.4, -1.4) = 0.43 \text{ failure}$$

$$x_1 = -1 - 0.6 = -1.6 \quad f(-1.6, -1.4) = 0.43 \text{ failure}$$

$$x_2 = -1.4 + 0.84 = -0.56 \quad f(-1, -0.56) = 3.18 \text{ success}$$

$$x^{(5)} = (-1, -0.56) .$$

To determine if the pattern search B was a success, $f(-1, -0.56) = 3.18$ is compared to $f(0.20,0.28) = 0.67$. Therefore, the pattern search is a success as $3.18 > 0.67$ and the new base point is

$$x^{(5)} = (-1, -0.56) \rightarrow f(-1, -0.56) = 3.18 .$$

and the old base point is

$$x^{(3)} = (0.20, 0.28) \rightarrow f(0.20, 0.28) = 0.67 \quad .$$

$C \rightarrow$ Since success has occurred, we proceed with another pattern search.

$$x^{(6)} = 2(x^{(5)}) - x^{(3)}$$

$$x^{(6)} = 2(-1, -0.56) - (0.20, 0.28)$$

$$x^{(6)} = (-2.20, -1.4) \rightarrow f(-2.2, -1.4) = 0.29 \quad .$$

A type II exploratory search is now performed on this new point to determine if the pattern move was a success.

Variable ϵ_i

$$x_1 = -2.20 + 0.60 = -1.60 \rightarrow f(-1.6, -1.40) = 0.43 \text{ success}$$

$$x_2 = -1.40 + 0.84 = -0.56 \rightarrow f(-1.6, -0.56) = 1.49 \text{ success}$$

$$x^{(7)} = (-1.6, -0.56) \quad .$$

To determine if pattern search C was a success, $f(-1.6, -0.56) = 1.49$ is compared to $x^{(5)}$ (current base point) $= (-1, -0.56) \rightarrow f(x^{(5)}) = 3.18$. Therefore $1.49 < 3.18$, and the pattern search was a failure. Consequently we would go back to $x^{(5)} = (-1, -0.56)$ and again initiate a type I search. The restart occurred due to the failure of the pattern search even though there were successes in the $x^{(7)}$ type II search. When the stage is reached in which neither the type I exploratory search nor the pattern search (together with a type II search) has a success in any coordinate direction, both are said to fail and the perturbation ϵ_i is reduced. Bazaraa and Shetty suggest by a factor of 0.5 [46]. Therefore, a new ϵ_i would be $(0.60/2, 0.84/2)$ which is equal to $(0.30, 0.42)$. In this example the optimal solution ultimately converges to $x_1 = -1$, $x_2 = 0$, and $f(x) \rightarrow \infty$.

3. Improvements, Advantages, and Disadvantages

Two suggestions from Gottfried and Weisman [47] that can improve the algorithm are as follows:

1. The algorithm can be speeded up by "remembering" the direction of improvement for each x_i in the previous exploratory sequence and then applying the same strategy during current exploration. For example, if the function shows improvement by decreasing given x_i to $x_i - \epsilon_i$, then the next exploratory move should first decrease x_i to $x_i - \epsilon_i$ and then increase x_i to $x_i + \epsilon_i$ only if the move to $x_i - \epsilon_i$ was not a success.

2. Improvement also can be obtained if one varies the magnitude of each ϵ_i separately, depending on the history of successes or failures with respect to the exploratory moves in each direction.

The algorithm's main feature consists of following "ridges" and "valleys." The pattern move can take long steps in the assumed direction of valleys, and the exploratory moves find the way back to these valleys if a pattern move has climbed out of them. Although the method lacks mathematical elegance, it is a highly efficient optimization procedure. It is easily programmed and, as mentioned earlier, is particularly well suited to functions that exhibit a straight, sharp ridge or valley.

Before one regards this method as the ultimate in direct search techniques, it should be noted there is a shortcoming. The method may sometime fail to produce any further improvement in a function that contains tightly curved ridges or sharp-cornered contours while still far from a maximum or minimum [48].

III. PROPOSED ALGORITHM – AN IMPROVED EXPLORATORY SEARCH TECHNIQUE FOR INTEGER PROGRAMMING (IESIP)

A. Approach

1. General

By using the best features of the greedy procedure, neighborhood search, and the Hooke and Jeeves type I and type II exploratory searches, coupled with modifications and efficient rounding procedures, one can develop an improved algorithm that will often prove superior to the established branch-and-bound procedure. This new procedure will be described along with a flow diagram showing the steps involved in the solution process. Also described are the necessary modifications to existing procedures that make this new algorithm an improvement over others. Section B will describe some of the more intricate details of the procedure such as the rounding procedure and stopping rule.

2. Solution Procedure and Modifications Required to Existing Schemes

The IESIP procedure begins, just as does the fractional-cut and branch-and-bound methods, with a continuous optimal solution obtained from simplex (linear) problems. Next, the continuous solution is rounded to an integer value according to the procedure detailed in section B.1. Once the rounded integer solution is obtained and shown to be within the constraint limitations, the next step employs some of the ideas of both the Hooke and Jeeves procedure for nonlinear problems and the greedy procedure discussed earlier.

The IESIP algorithm begins by utilizing an idea borrowed from the Hooke and Jeeves procedure. The first step involved in the Hooke and Jeeves procedure is a univariate exploratory search around the starting point.

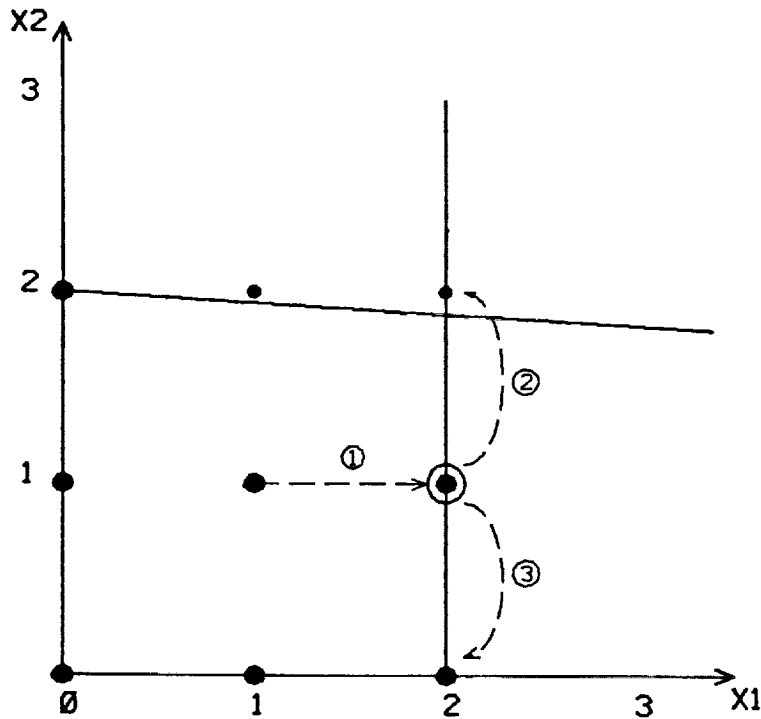


Figure 8. Hooke and Jeeves exploratory search progression.

Each time the value of the objective function increases (maximization problem) using this procedure, one keeps the value of the changed variable and moves on to the next variable, increasing it and then decreasing it to check for additional improvement. This results in the movement as shown in figure 8 for a two-variable problem. This idea works well with continuous type problems, but in the case of integer problems, this technique will tend to “miss” the optimal solution. This is especially apparent in the cases where the optimal integer solution is not as close to the continuous solution as one would expect. The following example shows the dilemma created on a two-variable problem where the Hooke and Jeeves exploratory search is applied directly to the problem without modifications:

$$\text{maximize } Z = f(x_1, x_2) = x_1 + 5x_2$$

$$\text{subject to: } x_1 + 10x_2 \leq 20$$

$$x_1 \leq 2$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

In applying the Hooke and Jeeves procedure a ϵ_i is needed to increment the variables in a univariate manner. Since this is an integer problem, the logical value for ϵ_i is an integer value of one or two if one starts with all integers in the solution. This is accomplished by first rounding the continuous solution of $f(2.0, 1.8) = 11$ to $f(1.0, 1.0) = 6$. The details of the rounding scheme employed and the justification for rounding to (1.0, 1.0) rather than (2.0, 1.0) is discussed later in section B.1. Type I exploratory search yields

<u>Variable</u>	ϵ_i	
$x_1 = 1 + 1 = 2$		$f(2,1) = 7$ (success)
$x_2 = 1 + 1 = 2$		$f(2,2) =$ constraint failure
$x_2 = 1 - 1 = 0$		$f(2,0) = 2$ (failure)

Therefore, the new base point is (2,1).

Pattern Search Fails

Type I search is performed again from (2,1). It also fails; therefore, with strict application, we reach a false optimum of $f(2,1) = 7$, instead of $f(0,2) = 10$, the actual integer optimum. Note that the x_1 value of 2 is kept once it meets successes at a value of 7, thereby not allowing the true solution to emerge as (0,2).

In order to overcome this problem, we must modify the exploratory search so that it does not rule out certain increasing (maximization) directions. This is accomplished by taking an idea from the greedy procedure and modifying it. Instead of selecting just the best alternative in our type I exploratory search, we will first change the ϵ value from a continuous value to an integer value of one. We then univariately examine the neighborhood around the rounded feasible solution to determine all feasible candidate integer solutions. Using the same example as shown above, this gives exploratory search results as shown below and in figure 9.

<u>Variable</u>	ϵ_i	
$x_1 = 1 + 1 = 2$		$f(2,1) = 7$ (feasible)
$x_1 = 1 - 1 = 0$		$f(0,1) = 5$ (feasible)
$x_2 = 1 + 1 = 2$		$f(1,2) =$ constraint failure
$x_2 = 1 - 1 = 0$		$f(1,0) = 1$ (feasible)

This then allows for type II exploration of each of these candidate feasible points. The type II exploration is similar to the type I in its univariate nature, but after examining all the feasible solutions from the type I search, only the best (largest objective function value for a maximization problem) is chosen to be examined further. In other words, each neighborhood of each feasible region is examined and only the best of all solutions from all neighborhoods is chosen to be examined further.

Each time a variable is incremented or decremented, the solution must be evaluated against the set of constraints. If one constraint fails, that variable is perturbed by a value of "1" in the other direction or the algorithm moves on to the next variable, if both directions have been

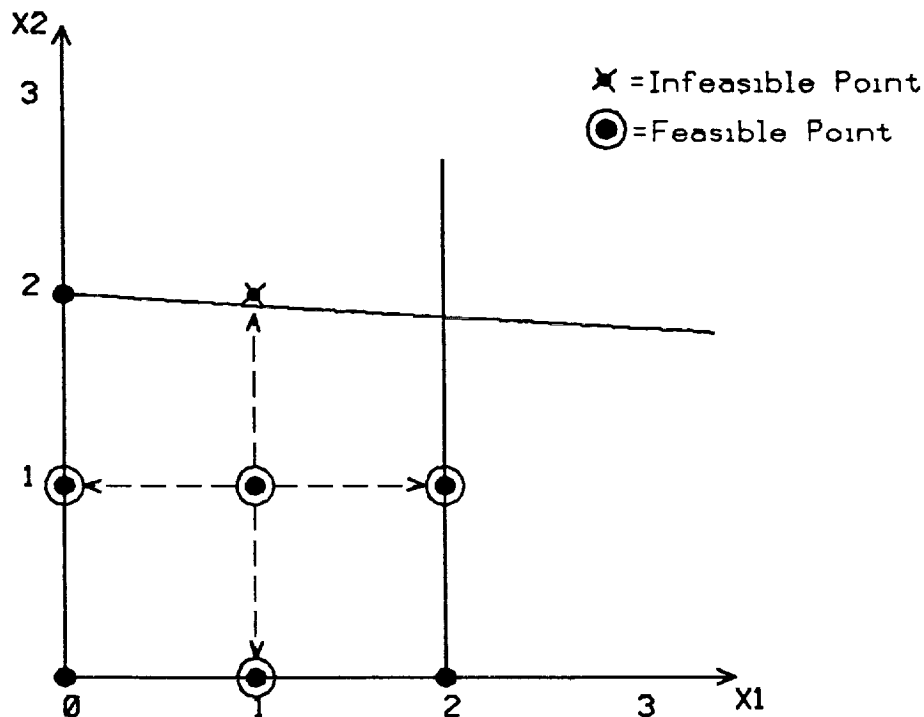


Figure 9. Modified type I exploratory search.

examined. This modification to the Hooke and Jeeves type I search was necessary to accommodate integer programming problems which are by their nature constrained. The Hooke and Jeeves procedure was originally designed for use on nonlinear unconstrained problems.

Another modification pertains to the order involved in incrementing and decrementing the variable. For maximization problems, the algorithm will first increment the variable by "1" and check the constraints, then it will decrement the variable by "1," check constraints, and move on to the next variable. For minimization problems, the algorithm will first decrement the variable by "1" and check constraints, then increment the variable by "1," check the constraints, and move on to the next variable.

This order involved in checking variables is important for the following reason: if all the variables in the objective function have positive coefficients, then the value obtained by the +1 increment with maximization or -1 decrement in minimization, will always be better than the succeeding -1 and +1 changes, respectively. Therefore, if the problem has all positive objective function coefficients, then the algorithm proceeds on to the next variable without checking the second increment (+ min, - max) if the first incrementation was successful. This computation saving scheme, of course, is only valid if used for type II searches. Type I searches identify all feasible candidates, both those obtained from +1 and -1 operations, disregarding the coefficients of the objective function. This computation saving feature can potentially save a tremendous amount of time, especially if applied to problems containing many variables.

Once the best solution has been obtained from all the feasible neighborhoods, a type II search is applied. Each time, the best solution is picked from this type II search and searched for

further improvement. This is continued until the "best" solution from a type II search is the same as the solution which initiated that particular search. At this point all the other feasible points on this neighborhood search are also examined. If their examination reveals a point that shows improvement over their respective feasible points, then it is examined by a type II search also. If not, then the algorithm terminates with the optimal solution being the current best value.

The following definitions, procedure outline, and flow diagram (fig. 10) describe the basic steps involved in the improved exploratory search technique:

Definitions

Type I Exploratory Search: This utilizes the best features of the greedy procedure and the Hooke and Jeeves type I exploratory search by combining them into one search scheme. This search produces a list of feasible solutions from the initial rounded integer point.

Type II Exploratory Search: This is similar to the type I search except that only the best solution is kept for further searching rather than all feasible points.

Parent Point: This defines a best point in a search, that if chosen to be examined will generate a sequence which returns to the present point. In other words, it is a "parent" to the current search and will only loop back on itself.

Seed Point: This defines the point that is currently being searched or explored.

Solution Steps for IESIP

Step 1: Round the optimal continuous solution to a feasible integer solution. Verify that the solution meets constraints; if it does not, then the term with the highest coefficient in the failed constraint is reduced by a value of one (maximization).

Step 2: Initiate the type I exploratory search to determine all feasible candidates for further examination.

Step 3: Initiate type II exploratory search of each feasible point found in step 2.

Step 4: Take the best solution(s) of all these searches and initiate another type II exploratory search on this point or points.

Step 5: Taking the best solution obtained in this search, check to see if it (they) is (are) a "parent" point(s). If not, go back to step 4. If yes, then proceed to step 6.

Step 6: Check to see if this is the first occurrence of a parent point; if it is, then go to step 7. If it is not, then choose the largest (maximization problem) nonparent point value that is an improvement over its respective seed point (the first seed point being the rounded continuous optimal). If there are none, then stop—the best point so far is optimal.

Step 7: Perform a type II search on all remaining feasible nonparent points. If any of these results is an improvement over its respective feasible points and is not a parent point, then go back to step 4. If there are none, then stop—the best point so far is optimal.

Note: In step 6, if there are multiple parent occurrences in multiple searches, then only the feasible nonparent solutions from the searches that contain a parent point are examined.

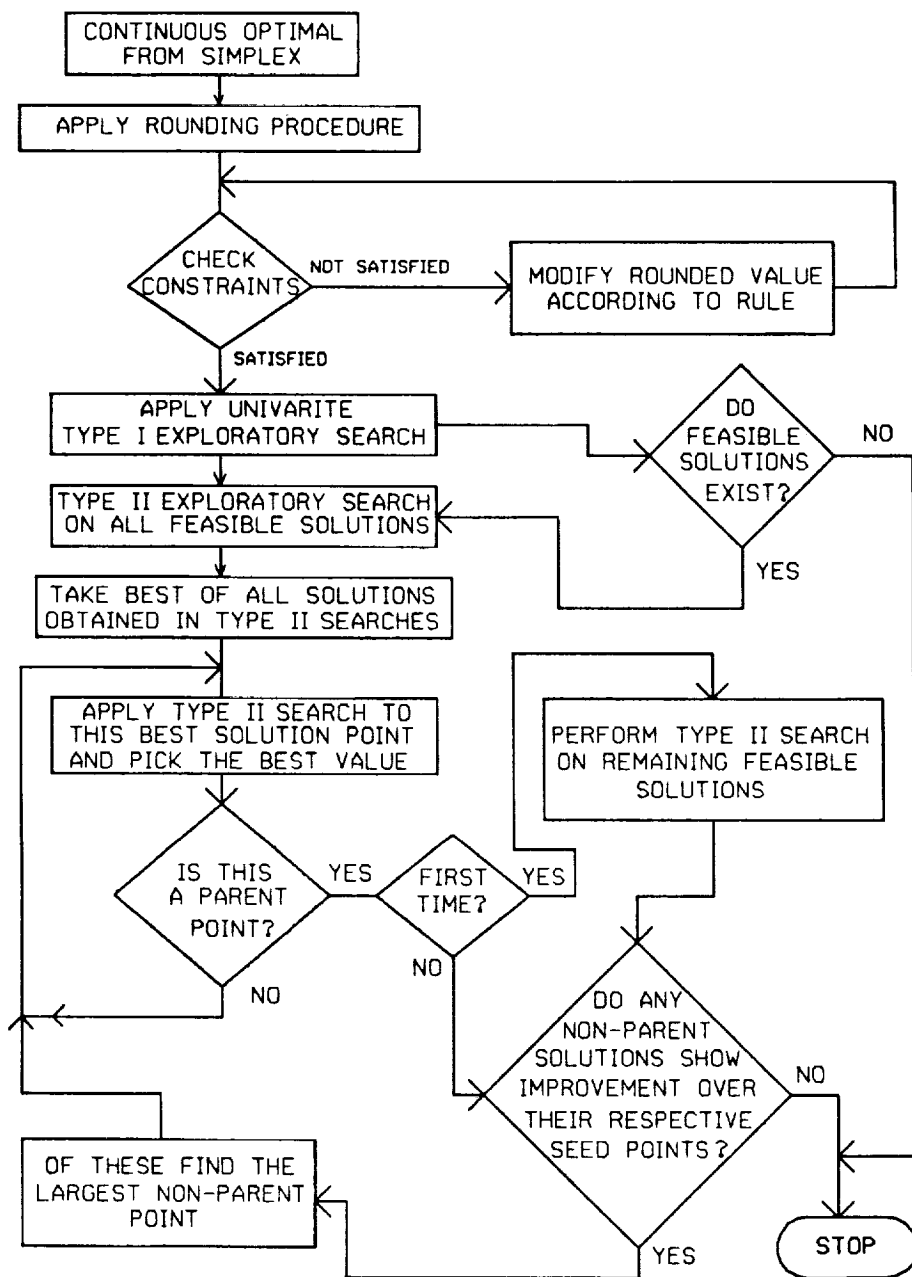


Figure 10. IESIP descriptive flow diagram.

3. Advantages in Using the Proposed Algorithm

The most significant advantage of this procedure over other integer programming techniques is that it primarily involves adding and subtracting "1" from the elements of an n -dimensional vector. The only other arithmetic operation that is necessary is multiplying the candidate solution vectors by the respective coefficients in the constraints and objective function.

Adding to this procedure's list of advantages is its lack of having to solve simplex iterations. When using a fractional-cut or branch-and-bound technique, one has to solve a new simplex problem each time a new constraint is added to the tableau. From a computational complexity standpoint, an iteration of dual simplex with four or five variables and four or five constraints is a laborious nightmare. This procedure eliminates that task completely.

Another advantage of this procedure is that it employs the better aspects of a proven non-linear optimization technique (Hooke and Jeeves) and a well-established heuristic procedure (greedy). It is further improved by the fact that the type I search avoids the greedy procedure pitfall of taking only the one best solution in a search. The following example shows how if a greedy procedure were applied in place of the modified type I exploratory search, then the resulting solution would not be optimal:

$$\text{maximize } Z = f(x_1, x_2) = 4,000x_1 + 7,000x_2$$

$$\text{subject to: } 1,200x_1 + 2,000x_2 \leq 6,000$$

$$25,000x_1 + 80,000x_2 \leq 200,000$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

$$\text{Rounded Solution} \rightarrow f(1,1) = 11,000$$

Type I exploratory search on (1,1) yields

Variable ϵ_j

$$x_1 = 1 + 1 = 2 \quad f(2,1) = 15,000 \text{ (feasible)}$$

$$x_1 = 1 - 1 = 0 \quad f(0,1) = 7,000 \text{ (feasible)}$$

$$x_2 = 1 + 1 = 2 \quad f(1,2) = 18,000 \text{ (feasible)}$$

$$x_2 = 1 - 1 = 0 \quad f(1,0) = 4,000 \text{ (feasible)}$$

Therefore, the feasible candidate solutions are (2,1), (0,1), (1,2), and (1,0). If we applied an unmodified greedy procedure, we would pick $f(1,2) = 18,000$ and examine it further. This would give the following:

<u>Variable</u>	ϵ_i	
$x_1 = 1 + 1 = 2$		$f(2,2) =$ fails constraint
$x_1 = 1 - 1 = 0$		$f(0,2) = 14,000$ (feasible)
$x_2 = 2 + 1 = 3$		$f(1,3) =$ fails constraint
$x_2 = 2 - 1 = 1$		$f(1,1) = 11,000$ (feasible)

Choosing the best solution yields $f(0,2) = 14,000$. Examining the point (0,2), we have

<u>Variable</u>	ϵ_i	
$x_1 = 0 + 1 = 1$		$f(1,2) = 18,000$ (feasible)
$x_2 = 2 + 1 = 3$		$f(0,3) =$ fails constraint
$x_2 = 2 - 1 = 1$		$f(0,1) = 7,000$ (feasible)

The largest value is $f(1,2) = 18,000$, but this is the first parent point (i.e., if chosen it will generate a sequence which returns to the present point). Therefore, our rule says to examine all other nonparent point feasible solutions of this search. The point $f(0,1) = 7,000$ is the only remaining point; therefore, it is examined.

<u>Variable</u>	ϵ_i	
$x_1 = 0 + 1 = 1$		$f(1,1) = 11,000$ (feasible)
$x_2 = 1 + 1 = 2$		$f(0,2) = 14,000$ (feasible)
$x_2 = 1 - 1 = 0$		$f(0,0) = 0$ (feasible)

Both (1,1) and (0,2) are parent points and $f(0,0) = 0$ is not an improvement over 7,000; therefore, we would conclude the procedure at this point with a solution of $f(1,2) = 18,000$ (best so far). This is not optimal. If we apply the type I exploratory search as described, we then would examine all three feasible candidates. An examination of the feasible point (2,1) gives a best solution of $f(3,1) = 19,000$, and after two more type II explorations, this point is improved again to an optimal of $f(5,0) = 20,000$. This example is shown in more detail in the next part of this section.

B. Other Details of the Algorithm

1. Rounding to Discrete Starting Solution

Considering the difficulties encountered with rounding continuous optimal solutions to integer values, this procedure will utilize the simple approach as discussed in section II by Baum and Trotter [35]. A rounded integer value is required of this algorithm since all of its subsequent steps require integer values and manipulations. Wismer and Chattergy in their discussion of rounding suggest that rounding is especially applicable when the optimal solution will be insensitive to a change of plus or minus one, e.g., when the solution is a large number [49]. This algorithm uses the idea of Baum and Trotter and extends it to cases where constraints are involved.

The rounding step of the algorithm proceeds in the following manner: first, in a maximization problem, the continuous optimal solution will be rounded down to the next lowest integer values. If a value is already an integer, it will be rounded down by an increment of one to the next lower integer value (e.g., (6.3,4) would be rounded down to (5,3)). In some of the test problems false optima were encountered when the integer values in continuous solutions were not reduced (or raised) to the next lower (or higher) increment of "1." Secondly, a constraint check is performed to verify that the rounded value falls within the constraints. If a rounded point fails a constraint, then the entry with the highest coefficient in the failed constraint is lowered by a value of 1. The procedure follows the same line of reasoning when dealing with a minimization problem except that the rounding will be up instead of down and the variable will be raised by "1" on a constraint failure rather than lowered.

This approach works well with maximization problems containing positive decision variables and " \leq " type constraints, or on minimization problems with " \geq " type constraints. Problems not in these forms will sometimes require extra effort in coming up with a rounded starting solution. The computer program presented at the end of this section will allow the user to round the continuous optimal solution independent of the program if so desired.

"Equal to" constraints are a problem for rounding in that they require a unique solution to even feasibly start the procedure. The problems of "equal to" constraints will be examined in section V when discussing restrictions of the algorithm.

2. Constraint Involvement

Integer programming problems, by their very nature, involve constraints or restrictions on their solutions. The Hooke and Jeeves exploratory search was designed for unconstrained problems; therefore, a step had to be included in IESIP for verifying that the candidate solution meets the constraints. Some of these problems are easy to check, but others are more laborious. For instance, when exploring around the point "0," and by using a value of -1 , we automatically trigger a constraint failure since this violates the non-negativity requirement found in almost all integer programming problems. However, for other points, the algorithm must accommodate a left-hand-side constraint value calculation at the check point with a comparison to the respective right-hand-side. If it passes the constraint requirement, the algorithm moves on to the next constraint. If at any time the constraint check fails, then the candidate solution at those points is said to fail. These constraint checks are conducted after every univariate search step performed on a variable and during the rounding procedure. Therefore, with this in mind, it is easy to see that the computational complexity and time required to utilize this algorithm is going to largely depend on the number of constraints involved in a particular problem.

3. Stopping Rule

The algorithm stops when the only points produced from a search are parent points together with points that are constraint failure points or points that show no improvement over the current seed value. The key to determining when this occurs is the recognition and storage of the "parent" points that have been previously examined, and knowledge of the origin of the present point being examined (i.e., did it come from one of the parent points?).

In the example presented in section C, the following initial rounded solution will give the associated feasible points after a type I exploratory search:

$$\begin{aligned} \text{Initial} \longrightarrow & f(1,1) = 11,000 \\ & f(2,1) = 15,000 \text{ (feasible)} \\ & f(0,1) = 7,000 \text{ (feasible)} \\ & f(1,2) = 18,000 \text{ (feasible)} \\ & f(1,0) = 4,000 \text{ (feasible)} \end{aligned}$$

After examining these feasible points with a type II search, the best solution came from examining (2,1) and was (3,1) with an objective function value of 19,000. Therefore, we say that (2,1) was the parent point for the current best solution of (3,1). Going further, we find that if the neighborhood around (3,1) is examined, it reveals the following:

$$\begin{aligned} & f(4,1) = \text{constraint failure} \\ & f(2,1) = 15,000 \text{ (feasible)} \\ & f(3,2) = \text{constraint failure} \\ & f(3,0) = 12,000 \text{ (feasible)} \end{aligned}$$

The point (2,1) is the best solution from this search, but it is recognized as the parent to the current search and is therefore not examined. Examination of this point would generate a sequence which returns to the present point. This is called by the procedure the "first criterion for stopping."

Also shown as a feasible point is the point $f(3,0) = 12,000$. The procedure therefore examines the neighborhood around this point for improvement over it. If there had been other feasible points besides (3,0), they would also have been examined.

$$\text{Searching } f(3,0) = 12,000 \text{ yields}$$

$$\underline{\text{Variable}} \quad \epsilon_i$$

$$\begin{aligned} x_1 = 3 + 1 = 4 & \quad f(4,0) = 16,000 \text{ (feasible)} \\ x_2 = 0 + 1 = 1 & \quad f(3,1) = 19,000 \text{ (feasible)} \\ x_2 = 0 - 1 = -1 & \quad f(3,+1) = \text{constraint failure} \end{aligned}$$

If the procedure is fortunate and there are no other feasible points other than the "parent-loop" point, then the algorithm ends here with the current best solution as the answer. Therefore, in the example above $f(4,0) = 16,000$ is examined (since $f(3,1) = 19,000$ is a parent point) revealing the following:

$$\underline{\text{Variable}} \quad \epsilon_i \quad f(4,0) = 16,000$$

$$\begin{aligned} x_1 = 4 + 1 = 5 & \quad f(5,0) = 20,000 \text{ (feasible)} \\ x_2 = 0 + 1 = 1 & \quad f(4,1) = \text{constraint failure} \\ x_2 = 0 - 1 = -1 & \quad f(4,-1) = \text{constraint failure} \end{aligned}$$

This allows us to pursue the point (5,0) since it is the best value and not a parent-loop point. Upon examining (5,0), we find the following:

<u>Variable</u>	ϵ_i	$f(5,0) = 20,000$
$x_1 = 5 + 1 = 6$		$f(6,0) = \text{constraint failure}$
$x_1 = 5 - 1 = 4$		$f(4,0) = 16,000$
$x_2 = 0 + 1 = 1$		$f(5,1) = \text{constraint failure}$
$x_2 = 0 - 1 = -1$		$f(5,-1) = \text{constraint failure}$

Since (4,0) is the only feasible point and it is the parent point for the current search, the algorithm has no other feasible points to examine and therefore stops. It should also be noted that more than one parent point could be found in a search and would also not need to be reexamined.

In summary, the stopping rule consists of the following two criteria:

1. The best solutions from a search are parents to the current search and will therefore result in a loop-back on themselves if examined.
2. If criterion (1) is met and there are no other feasible points in that search that are improvements over their current seed point, then the procedure stops. The current best solution is chosen as the optimal.

C. Example Problems

The following example is presented in detail to illustrate the procedure. It is a particularly interesting problem in that it is one of the class of problems where the optimal integer solution (5,0) is relatively far removed from its optimal continuous solution of (1.739,1.956). The example is taken from Claycombe and Sullivan [50]. The first part of the example solves the problem using the IESIP procedure. Following the solution is a graphic representation of the solution process showing its progress from rounded initial interger solution to the final optimal value. Another example with four variables is also provided after the graphic.

Two Variable Example

An excursion company is considering adding small boats to its fleet. The company has \$200,000 to invest in this venture. At present, there is an estimated maximum demand of 6,000 customers per season for these tours. The company does not wish to provide capacity in excess of the esimated maximum demand. The basic data are given below for the two types of available boats. The company will make an estimated seasonal profit of \$4,000 for each boat of type A and \$7,000 for each boat of type B. How many boats of each type should it purchase?

	<u>Type A</u>	<u>Type B</u>
Capacity (customers/season)	1,200	2,000
Initial Cost (\$/boat)	25,000	80,000

This translates into the following integer linear programming problem:

$$\text{maximize } Z = f(x_1, x_2) = 4,000x_1 + 7,000x_2$$

$$\text{subject to: } 1,200x_1 + 2,000x_2 \leq 6,000$$

$$25,000x_1 + 80,000x_2 \leq 200,000$$

$$x_1, x_2 \geq 0 \text{ and integer .}$$

Step 1: The continuous optimal solution is given as (1.739, 1.956) which gives a value of \$20,648 to the objective function. Applying the rounding rules to this value, we have a rounded solution of (1, 1) = \$11,000. We then check this rounded value against the constraints and find it to be feasible.

Step 2: Perform a type I exploratory search around the rounded initial solution. After each increment or decrement of a variable, the constraints are checked to verify feasibility.

<u>Variable</u>	ϵ_i	$f(1,1) = 11,000$
$x_1 = 1 + 1 = 2$		$f(2,1) = 15,000$ (feasible)
$x_1 = 1 - 1 = 0$		$f(0,1) = 7,000$ (feasible)
$x_2 = 1 + 1 = 2$		$f(1,2) = 18,000$ (feasible)
$x_2 = 1 - 1 = 0$		$f(1,0) = 4,000$ (feasible)

All points are feasible; therefore, the next step is to proceed on to the type II examination of each of the neighborhoods around these feasible points.

Step 3: Examine the neighborhoods around each of the feasible points with a type II exploratory search. After each increment or decrement of a variable, the constraints are checked to verify feasibility.

<u>Variable</u>	ϵ_i	$f(2,1) = 15,000$
$x_1 = 2 + 1 = 3$		$f(3,1) = 19,000$ (feasible)*
$x_2 = 1 + 1 = 2$		$f(2,2) = \text{constraint failure}$
$x_2 = 1 - 1 = 0$		$f(2,0) = 8,000$ (feasible)

<u>Variable</u>	ϵ_i	$f(0,1) = 7,000$
$x_1 = 0 + 1 = 1$		$f(1,1) = 11,000$ (feasible)
$x_2 = 1 + 1 = 2$		$f(0,2) = 14,000$ (feasible)

<u>Variable</u>	ϵ_i	$f(1,2) = 18,000$
$x_1 = 1 + 1 = 2$		$f(2,2) = \text{constraint failure}$
$x_1 = 1 - 1 = 0$		$f(0,2) = 14,000$ (feasible)
$x_2 = 2 + 1 = 3$		$f(1,3) = \text{constraint failure}$
$x_2 = 2 - 1 = 1$		$f(1,1) = 11,000$ (feasible)

<u>Variable</u>	ϵ_i	$f(1,0) = 4,000$
$x_1 = 1 + 1 = 2$		$f(2,0) = 8,000$ (feasible)
$x_2 = 0 + 1 = 1$		$f(1,1) = 11,000$ (feasible)

Step 4: Take the best solution point found in step 3 (*) and initiate a type II exploratory search on this point.

<u>Variable</u>	ϵ_i	$f(3,1) = 19,000$
$x_1 = 3 + 1 = 4$		$f(4,1) = \text{constraint failure}$
$x_1 = 3 - 1 = 2$		$f(2,1) = 15,000$ (feasible)
$x_2 = 1 + 1 = 2$		$f(3,2) = \text{constraint failure}$
$x_2 = 1 - 1 = 0$		$f(3,0) = 12,000$ (feasible)

Step 5: Check to see if the best solution obtained in step 4 is the same as the solution which initiated that search (parent point). If not, proceed to step 4 again. If yes, go to step 6.

Step 6: In this case, (2,1) is the parent point and the first occurrence of such a point. Therefore, the remaining feasible point is only $f(3,0) = 12,000$.

<u>Variable</u>	ϵ_i	$f(3,0) = 12,000$
$x_1 = 3 + 1 = 4$		$f(4,0) = 16,000$ (feasible)
$x_1 = 3 - 1 = 2$		$f(2,0) = 8,000$ (feasible)
$x_2 = 0 + 1 = 1$		$f(3,1) = 19,000$ (feasible)
$x_2 = 0 - 1 = -1$		$f(3,-1) = \text{constraint failure}$

Step 7: Perform type II search on each of the other feasible points. If any of these results is an improvement over its respective feasible points and is not a parent point, go back to step 4. If there is none, then stop—the current best is the solution. The best point is (3,1), is a parent point, and should not be examined.

In this case, the point (4,0) is the next best point and an improvement over 12,000, and it is also not a parent point. Therefore, step 4 is performed again on the new point (4,0).

<u>Variable</u>	ϵ_i	$f(4,0) = 16,000$
$x_1 = 4 + 1 = 5$		$f(5,0) = 20,000$ (feasible)
$x_2 = 0 + 1 = 1$		$f(4,1) = \text{constraint failure}$
$x_2 = 0 - 1 = -1$		$f(4,-1) = \text{constraint failure}$

Best point is $f(5,0) = 20,000$ and not a parent point. Step 4 is invoked again.

<u>Variable</u>	ϵ_i	$f(5,0) = 20,000$
$x_1 = 5 + 1 = 6$		$f(6,0) = \text{constraint failure}$
$x_1 = 5 - 1 = 4$		$f(4,0) = 16,000$ (feasible)
$x_2 = 0 + 1 = 1$		$f(5,1) = \text{constraint failure}$
$x_2 = 0 - 1 = -1$		$f(5,-1) = \text{constraint failure}$.

(4,0) is the best and only feasible point at this search. According to step 6, since this is a parent point and there are no other improving feasible points from this search, the solution is the best one so far; that is, $f(5,0) = 20,000$. This example is shown graphically in figure 11.

Four Variable Example

$$\text{maximize } Z = f(x_1, x_2, x_3, x_4) = 4x_1 + 5x_2 + 9x_3 + 5x_4$$

$$\text{subject to: } x_1 + 3x_2 + 9x_3 + 6x_4 \leq 16$$

$$6x_1 + 6x_2 + 7x_4 \leq 19$$

$$7x_1 + 8x_2 + 18x_3 + 3x_4 \leq 44$$

$$x_1, x_2, x_3, x_4 \geq 0 \text{ and integer .}$$

Continuous Optimal Solution = $f(2.8652, 0, 1.2871, 0.2584) = 24.337$.

Step 1: Rounded feasible starting point = $f(2, 0, 1, 0) = 17$

Step 2:

<u>Variable</u>	ϵ_i	$f(2,0,1,0) = 17$
$x_1 = 2 + 1 = 3$		$f(3,0,1,0) = 21$ (feasible)
$x_1 = 2 - 1 = 1$		$f(1,0,1,0) = 13$ (feasible)
$x_2 = 0 + 1 = 1$		$f(2,1,1,0) = 22$ (feasible)
$x_2 = 0 - 1 = -1$		$f(2,-1,1,0) = \text{constraint failure}$
$x_3 = 1 + 1 = 2$		$f(2,0,2,0) = \text{constraint failure}$
$x_3 = 1 - 1 = 0$		$f(2,0,0,0) = 8$ (feasible)
$x_4 = 0 + 1 = 1$		$f(2,0,1,1) = \text{constraint failure}$
$x_4 = 0 - 1 = -1$		$f(2,0,1,-1) = \text{constraint failure}$.

Four feasible points are now examined by step 3.

Step 3:

<u>Variable</u>	ϵ_i	$f(3,0,1,0) = 21$
$x_1 = 3 + 1 = 4$		$f(4,0,1,0) =$ constraint failure
$x_1 = 3 - 1 = 2$		$f(2,0,1,0) = 17$ (feasible)
$x_2 = 0 + 1 = 1$		$f(3,1,1,0) =$ constraint failure
$x_2 = 0 - 1 = -1$		$f(3,-1,1,0) =$ constraint failure
$x_3 = 1 + 1 = 2$		$f(3,0,2,0) =$ constraint failure
$x_3 = 1 - 1 = 0$		$f(3,0,0,0) = 12$ (feasible)
$x_4 = 0 + 1 = 1$		$f(3,0,1,1) =$ constraint failure
$x_4 = 0 - 1 = -1$		$f(3,0,1,-1) =$ constraint failure .

<u>Variable</u>	ϵ_i	$f(1,0,1,0) = 13$
$x_1 = 1 + 1 = 2$		$f(2,0,1,0) = 17$ (feasible)
$x_1 = 1 - 1 = 0$		$f(0,0,1,0) = 9$ (feasible)
$x_2 = 0 + 1 = 1$		$f(1,1,1,0) = 18$ (feasible)
$x_2 = 0 - 1 = -1$		$f(1,-1,1,0) =$ constraint failure
$x_3 = 1 + 1 = 2$		$f(1,0,2,0) =$ constraint failure
$x_3 = 1 - 1 = 0$		$f(1,0,0,0) = 4$ (feasible)
$x_4 = 0 + 1 = 1$		$f(1,0,1,1) = 18$ (feasible)
$x_4 = 0 - 1 = -1$		$f(1,0,1,-1) =$ constraint failure .

<u>Variable</u>	ϵ_i	$f(2,1,1,0) = 22$
$x_1 = 2 + 1 = 3$		$f(3,1,1,0) =$ constraint failure
$x_1 = 2 - 1 = 1$		$f(1,1,1,0) = 18$ (feasible)
$x_2 = 1 + 1 = 2$		$f(2,2,1,0) =$ constraint failure
$x_2 = 1 - 1 = 0$		$f(2,0,1,0) = 17$ (feasible)
$x_3 = 1 + 1 = 2$		$f(2,1,2,0) =$ constraint failure
$x_3 = 1 - 1 = 0$		$f(2,1,0,0) = 13$ (feasible)
$x_4 = 0 + 1 = 1$		$f(2,1,1,1) =$ constraint failure
$x_4 = 0 - 1 = -1$		$f(1,0,1,-1) =$ constraint failure .

<u>Variable</u>	ϵ_i	$f(2,0,0,0) = 8$
$x_1 = 2 + 1 = 3$		$f(3,0,0,0) = 12$ (feasible)
$x_1 = 2 - 1 = 1$		$f(1,0,0,0) = 4$ (feasible)
$x_2 = 0 + 1 = 1$		$f(2,1,0,0) = 13$ (feasible)
$x_2 = 0 - 1 = -1$		$f(2,-1,0,0) =$ constraint failure
$x_3 = 0 + 1 = 1$		$f(2,0,1,0) = 17$ (feasible)
$x_3 = 0 - 1 = -1$		$f(2,0,-1,0) =$ constraint failure
$x_4 = 0 + 1 = 1$		$f(2,0,0,1) = 13$ (feasible)
$x_4 = 0 - 1 = -1$		$f(2,0,0,-1) =$ constraint failure .

The best solutions of all neighborhoods examined above are the points (1,1,1,0) and (1,0,1,1), both of which result in objective function values of 18.

<u>Variable</u>	ϵ_i	$f(1,1,1,0) = 18$
$x_1 = 1 + 1 = 2$		$f(2,1,1,0) = 22$ (feasible)
$x_1 = 1 - 1 = 0$		$f(0,1,1,0) = 12$ (feasible)
$x_2 = 1 + 1 = 2$		$f(1,2,1,0) = 23$ (feasible)
$x_2 = 1 - 1 = 0$		$f(1,0,1,0) = 13$ (feasible)
$x_3 = 1 + 1 = 2$		$f(1,1,2,0) =$ constraint failure
$x_3 = 1 - 1 = 0$		$f(1,1,0,0) = 9$ (feasible)
$x_4 = 0 + 1 = 1$		$f(1,1,1,1) =$ constraint failure
$x_4 = 0 - 1 = -1$		$f(1,1,1,-1) =$ constraint failure .

<u>Variable</u>	ϵ_i	$f(1,0,1,1) = 18$
$x_1 = 1 + 1 = 2$		$f(2,0,1,1) =$ constraint failure
$x_1 = 1 - 1 = 0$		$f(0,0,1,1) = 14$ (feasible)
$x_2 = 0 + 1 = 1$		$f(1,1,1,1) =$ constraint failure
$x_2 = 0 - 1 = -1$		$f(1,-1,1,1) =$ constraint failure
$x_3 = 1 + 1 = 2$		$f(1,0,2,1) =$ constraint failure
$x_3 = 1 - 1 = 0$		$f(1,0,0,1) = 9$ (feasible)
$x_4 = 1 + 1 = 2$		$f(1,0,1,2) =$ constraint failure
$x_4 = 1 - 1 = 0$		$f(1,0,1,0) = 13$ (feasible) .

The best point found in all these neighborhoods is $f(1,2,1,0) = 23$, and it is not a parent point.

<u>Variable</u>	ϵ_i	$f(1,2,1,0) = 23$
$x_1 = 1 + 1 = 2$		$f(2,2,1,0) =$ constraint failure
$x_1 = 1 - 1 = 0$		$f(0,2,1,0) = 19$ (feasible)*
$x_2 = 2 + 1 = 3$		$f(1,3,1,0) =$ constraint failure
$x_2 = 2 - 1 = 1$		$f(1,1,1,0) = 18$ (feasible)
$x_3 = 1 + 1 = 2$		$f(1,2,2,0) =$ constraint failure
$x_3 = 1 - 1 = 0$		$f(1,2,0,0) = 14$ (feasible)
$x_4 = 0 + 1 = 1$		$f(1,2,1,1) =$ constraint failure
$x_4 = 0 - 1 = -1$		$f(1,2,1,-1) =$ constraint failure .

The best point found in this search (*) is $f(0,2,1,0) = 19$. It is not a parent point so we examine it further.

<u>Variable</u>	ϵ_i	$f(0,2,1,0) = 19$
$x_1 = 0 + 1 = 1$		$f(1,2,1,0) = 23$ (feasible)*
$x_1 = 0 - 1 = -1$		$f(-1,2,1,0) =$ constraint failure
$x_2 = 2 + 1 = 3$		$f(0,3,1,0) =$ constraint failure
$x_2 = 2 - 1 = 1$		$f(0,1,1,0) = 14$ (feasible)

$$\begin{array}{ll}
x_3 = 1 + 1 = 2 & f(0,2,2,0) = \text{constraint failure} \\
x_3 = 1 - 1 = 0 & f(0,2,0,0) = 10 \text{ (feasible)} \\
x_4 = 0 + 1 = 1 & f(0,2,1,1) = \text{constraint failure} \\
x_4 = 0 - 1 = -1 & f(0,2,1,-1) = \text{constraint failure} .
\end{array}$$

The best point in this search (*) is $f(1,2,1,0) = 23$, which is the parent point for this search. Therefore, according to step 7, we start again with a type II search of the remaining feasible points that are not parent points (i.e., $(0,1,1,0)$ and $(0,2,0,0)$) and take the best of these two searches.

$$\begin{array}{ll}
\text{Variable} & \epsilon_i & f(0,1,1,0) = 14 \\
x_1 = 0 + 1 = 1 & & f(1,1,1,0) = 18 \text{ (feasible)} \\
x_1 = 0 - 1 = -1 & & f(-1,0,1,0) = \text{constraint failure} \\
x_2 = 1 + 1 = 2 & & f(0,2,1,0) = 19 \text{ (feasible)} \\
x_2 = 1 - 1 = 0 & & f(0,0,1,0) = 9 \text{ (feasible)} \\
x_3 = 1 + 1 = 2 & & f(0,1,2,0) = \text{constraint failure} \\
x_3 = 1 - 1 = 0 & & f(0,1,0,0) = 5 \text{ (feasible)} \\
x_4 = 0 + 1 = 1 & & f(0,1,1,1) = \text{constraint failure} \\
x_4 = 0 - 1 = -1 & & f(0,1,1,-1) = \text{constraint failure} .
\end{array}$$

$$\begin{array}{ll}
\text{Variable} & \epsilon_i & f(0,2,0,0) = 10 \\
x_1 = 0 + 1 = 1 & & f(1,2,0,0) = 14 \text{ (feasible)} \\
x_1 = 0 - 1 = -1 & & f(-1,2,0,0) = \text{constraint failure} \\
x_2 = 2 + 1 = 3 & & f(0,3,0,0) = 15 \text{ (feasible)} \\
x_2 = 2 - 1 = 1 & & f(0,1,0,0) = 5 \text{ (feasible)} \\
x_3 = 0 + 1 = 1 & & f(0,2,1,0) = 19 \text{ (feasible)} \\
x_3 = 0 - 1 = -1 & & f(0,2,-1,0) = \text{constraint failure} \\
x_4 = 0 + 1 = 1 & & f(0,2,0,1) = 15 \text{ (feasible)} \\
x_4 = 0 - 1 = -1 & & f(0,2,0,-1) = \text{constraint failure} .
\end{array}$$

$(1,1,1,0)$ and $(0,2,1,0)$ are both previous parent points. Hence, using step 6, the next best point that is an improvement over 10 and 14 is a tie between $f(0,3,0,0) = 15$ and $f(0,2,0,1) = 15$. Examining both of these we have:

$$\begin{array}{ll}
\text{Variable} & \epsilon_i & f(0,2,0,1) = 15 \\
x_1 = 0 + 1 = 1 & & f(1,2,0,1) = \text{constraint failure} \\
x_1 = 0 - 1 = -1 & & f(-1,2,0,1) = \text{constraint failure} \\
x_2 = 2 + 1 = 3 & & f(0,3,0,1) = \text{constraint failure} \\
x_2 = 2 - 1 = 1 & & f(0,1,0,1) = 10 \text{ (feasible)} \\
x_3 = 0 + 1 = 1 & & f(0,2,1,1) = \text{constraint failure} \\
x_3 = 0 - 1 = -1 & & f(0,2,-1,1) = \text{constraint failure} \\
x_4 = 1 + 1 = 2 & & f(0,2,0,2) = \text{constraint failure} \\
x_4 = 1 - 1 = 0 & & f(0,2,0,0) = 10 \text{ (feasible)} .
\end{array}$$

<u>Variable</u>	ϵ_i	$f(0,3,0,0) = 15$
$x_1 = 0 + 1 = 1$		$f(1,3,0,0) = \text{constraint failure}$
$x_1 = 0 - 1 = -1$		$f(-1,3,0,0) = \text{constraint failure}$
$x_2 = 3 + 1 = 4$		$f(0,4,0,0) = \text{constraint failure}$
$x_2 = 3 - 1 = 2$		$f(0,2,0,0) = 10$ (feasible)
$x_3 = 0 + 1 = 1$		$f(0,3,1,0) = \text{constraint failure}$
$x_3 = 0 - 1 = -1$		$f(0,3,-1,0) = \text{constraint failure}$
$x_4 = 0 + 1 = 1$		$f(0,3,0,1) = \text{constraint failure}$
$x_4 = 0 - 1 = -1$		$f(0,3,0,-1) = \text{constraint failure}$

No improvements are seen over the respective initial points ($10 < 15$), so the procedure terminates. The integer solution is the best one so far; that is, $f(1,2,1,0) = 23$.

It should be noted that this procedure required only 13 univariate exploratory searches. The branch-and-bound procedure required to solve this problem took 25 iterations. Furthermore, each branch-and-bound iteration is a simplex problem to be solved, one of which is extremely more laborious than one exploratory search used here. For example, IESIP requires only integer arithmetic in adding and subtracting one to each of the variable values, whereas simplex or dual simplex requires the use of real arithmetic and the manual manipulation of dealing with fractions if one is solving the problem by hand. This comparison between the computational complexity of the two methods will be examined further in section IV.

D. Justification of Optimality

In discussing direct search heuristic procedures, certain questions such as the following are repeatedly asked: "Under what conditions does this direct search heuristic converge to the solution?" "Once a solution is found, how good is it?" "What are the advantages, if any, of this procedure over other more established methods?"

Although several algorithms have been developed for the integer problem, some of them are not uniformly efficient from the computational standpoint, particularly as the size of the problem increases [5]. Both the fractional cut and branch-and-bound methods require the computer to deal with fractions. This is a major difficulty in integer programming. The effect of round-off error that results from the use of the digital computer for solving these problems can sometimes be significant. Thus, heuristic algorithms including approximation, direction search, and the procedure presented here are developed to deal with integer problems with strictly integer values. This eliminates dealing with fractions.

The algorithm presented here is a modified direct search method that draws its abilities and advantages from these modified and previously proven concepts. These are the greedy procedure, neighborhood search, and the exploratory search developed by Hooke and Jeeves. This algorithm does not pretend to be the panacea for all integer problems. Nor does it claim to always provide the optimal integer solution in every case where it is applied. What it does claim is that it will obtain in most cases the same "optimal" solution found by branch-and-bound with a reduction in

computational work required to do so. It will perform as do other heuristic methods and find a solution which is optimal in almost all the cases examined by this research. The exceptions to this are noted in section V.

Since this procedure is heuristic in nature, its validation will be strongly empirical. Therefore, in section IV, 45 test problems are examined with the branch-and-bound and IESIP techniques to substantiate the abilities and advantages of the new method.

E. Computer Implementation

In order to efficiently evaluate a number of test problems, the IESIP algorithm must be automated. For ease of use and lower cost, the personal computer environment was chosen as the instrument for handling this task. An adequate but efficient language had to be selected for this algorithm. BASIC was ruled out due to its inherent shortcomings related to the screen input of text strings. FORTRAN was discounted due to the unavailability and high cost of a compiler. Other languages were also considered, but since the author had some familiarity with Pascal, and its attributes seemed to accommodate the needs of the algorithm, it was chosen as the language for IESIP.

The IESIP program was written in Pascal due mainly to the requirements for dynamic memory allocation, pointers, and structures. The size of a problem, the number of variables, and the number of nodes that must be expanded will vary depending on the statement of the problem. With dynamic memory allocation, one can allocate the spaces required for a new node when the new node is encountered, as opposed to static memory allocation that would require us to allocate the maximum amount of memory at the start of the program.

Pointers are used to facilitate the movement of data. That is, the data structures are stored on the linked list with pointers so we can move the data from one list to another by just moving the pointer as opposed to moving the data.

There are two basic structures used by this program. One is the structure of a data record. That is, a data record will hold the coefficients of up to 30 variables and 3 extra variables—1 for the value when evaluated with the objective function, 1 for the evaluation of the constraint check, and 1 for the operand if the node is a constraint function. Also, the data record will contain pointers to the previous and next data record on the linked list. The second structure is a list record that will contain three pointers, one to each of the data records, and the previous and next list elements.

The program can be divided into the three major sections of initialization, problem evaluation, and end printout. The reader is referred to the program listing in appendix B for further details of the purpose and function of each subroutine.

Tables 1 and 2 present a sample input and output of the program. The sample problem presented in these tables is the solution to the four-variable problem presented earlier in this section.

Table 1. Sample input from IESIP computer program.

```
      IESIP
Is the objective function to be maximized?
[y]es/[n]o/[h]elp default = no Y
Do you have the functions in a data file?
[y]es/[n]o/[h]elp default = no Y
Enter the file? TEST1.DAT
```

```
Maximization problem
Number of variables = 2
```

```
Objective function:
+4000.00 X1 +7000.00 X2
```

```
Constraint function(s):
+1200.00 X1 +2000.00 X2 <= 6000.00
+25000.00 X1 +80000.00 X2 <= 200000.00
```

```
Continuous solution rounded value:
(1, 1)
```

```
Do you wish to accept this rounded solution?
[y]es/[n]o/[h]elp default = yes Y
Enter the number of initial expansions
Integer or 0 for help
1
```

```
Do you wish to see the nodes as they are expanded?
[y]es/[n]o/[h]elp default = no Y
```

One interesting feature of the program is that it allows for either manual input or data encoding in a data file called up by the program to be evaluated. The program also allows the user to decide how many initial type I exploratory searches (with all feasible points examined) are needed. The program calculates the initial rounded solution for the user and asks the user if it is suitable. The user can examine each point expansion or can only view the final results. The program also keeps a count of the exact number of additions, comparisons, and multiplications required to both round and find a solution. A computational complexity score is also shown at the end of the solution output. The significance and justification for this feature are discussed in section IV.

Table 2. Sample output from IESIP computer program.

Node Expansion (1)

(1, 1) = 11000.000
(2, 1) = 15000.000
(0, 1) = 7000.000
(1, 2) = 18000.000
(1, 0) = 4000.000

Node Expansion (2)

(1, 2) = 18000.000
(2, 2) = C.F.
(0, 2) = 14000.000
(1, 3) = C.F.
(1, 1) = 11000.000
Press RETURN to continue...

Node Expansion (3)

(2, 1) = 15000.000
(3, 1) = 19000.000
(1, 1) = 11000.000
(2, 2) = C.F.
(2, 0) = 8000.000
:
:
:

Node Expansion (9)

(5, 0) = 20000.000
(6, 0) = C.F.
(4, 0) = 16000.000
(5, 1) = C.F.

Do you wish to have the problem restated?
[Y]es/[n]o/[h]elp default = no Y

Best solution(s) found:

(5, 0) = 20000.000

Computation of the rounded optimal solution required:

Additions: 2 Multiplications: 4 Compares: 2

9 Points were expanded requiring

Additions: 87 Multiplications: 174 Compares: 46

Computational complexity score = 863.827

IV. COMPARISON OF RESULTS – COMPUTATIONAL EXPERIMENTS

A. Algorithm Performance Measures

1. Current Methods

Developing a new algorithm such as the one presented in the preceding section is relatively easy compared to the task of appraising the algorithm on its merits and benefits. One not only wants an algorithm that works but an algorithm that is efficient. The efficiency of an algorithm can be appraised by a variety of criteria. One such criterion is the execution time required by an algorithm to solve a particular problem as compared to another algorithm's time to solve the same problem. However, such a measure is strongly dependent upon the coded program, languages, and machine used to implement the algorithm. Thus, a difference in the program may not represent a significant change in the underlying algorithm but may, nevertheless, affect the speed of execution. Furthermore, if two programs are compared first on one machine and then another, the comparisons may lead to different conclusions. Therefore, while comparison of actual programs running on real computers is an important source of information, the results are inevitably affected by programming skill and machine characteristics. Ignizio [51] recognized the comparison of algorithms based on computation time as poor at best. He further suggested that if selections and comparisons are to be intelligently made, a systematic procedure for recording and presenting an algorithm's performance must be implemented. He suggested that this procedure consist of the following: (1) a standard reporting format which contains essential factors such as dates, computer used, language used, memory used, etc., and (2) a measurement standard consisting of computation time, internal and external storage requirements, problem size, and accuracy (i.e., how close the algorithm came to the actual optimal answer).

An alternative to comparing execution times is the use of established software metrics. These metrics measure the complexity of the source code. Most of the metrics incorporate easily computed properties of the source code, such as the number of operators and operands, the complexity of the control flow graph, and the number of parameters and global variables in routines. The approach taken is to compute a number, or set of numbers, that measures the complexity of the code. Halstead [52] developed a number of metrics that are widely used and easily computed from the properties of the source code. These properties include the total number of operators in a program, the total number of operands in the program, and the number of unique operators and operands. Another metric available and also widely used was developed by Thomas McCabe. McCabe's cyclomatic complexity measure [53] attempts to measure and control the number of basic paths through a program. He represents the software via a connected graph and computes a cyclomatic number (or score) V , where V is found by

$$V = E - n + 2p$$

and E is the number of edges, n is the number of nodes, and p is the number of connected components.

The basic underlying drawback to these two approaches being utilized in comparing two algorithms is that they are extremely dependent on the software coding itself. Different programmers or the same programmer on different days could produce different results. Thus, if using these methods solely to compare a candidate algorithm to an existing algorithm, one might be comparing the software of a highly skilled programmer against a moderately skilled one.

Another approach to algorithm performance lies in the arena of accuracy. Parker and Rardin [25] and Lee and Lee [53] suggest that when comparing a nonexact heuristic algorithm to an exact procedure, a comparison should be made by "performance ratio." This bases the performance of the challenger algorithm solely on the quality of the solution obtained. The performance ratio in question is

$$P_H(t) = \begin{cases} \frac{v(t)}{v_H(t)} & \text{if } t \text{ is a maximization problem} \\ \frac{v_H(t)}{v(t)} & \text{if } t \text{ is a minimization problem.} \end{cases}$$

$v_H(t)$ is the value of the solution returned by the nonexact heuristic procedure H on instance t and $v(t)$ is the value of an optimal solution at instance t_0 . Several points or drawbacks to this approach are also noted. One is that the ratios make sense only when the solution values are non-negative. Also, adding a constant to the objective function changes no solutions, but does impact the ratio (e.g., a maximization ratio of 57/55 adjusted by a constant of 2 yields 59/57, hence 57/55 \neq 59/57). This ratio only has merit if most of the heuristic solutions are different from the optimal solution. If the purpose is to show that a particular algorithm is better than an existing algorithm on a certain class of problem and the optimal solution is nearly always obtained using the heuristic procedures, then the accuracy performance ratio would only show it performed equally well, not better.

2. Proposed Alternative

This research has produced an algorithm that competes with an existing algorithm (branch-and-bound) in the solution of pure integer programming problems. The preceding section outlined some accepted procedures for comparing algorithms against one another. These procedures are well established and, in the correct application, are useful in determining the efficiency or lack of efficiency of a new algorithm. However, since the author developed the software for the IESIP procedure and is comparing it to an existing commercially available branch-and-bound algorithm Quantitative Systems for Business (QSB), the use of comparing run-times and software metrics is ruled out. Furthermore, since a primary objective of this research was to show that the new algorithm is better than branch-and-bound on a certain class of problems, the accuracy performance ratio approach is not useful due to its inability to show improvement of the heuristic over the existing procedure.

With this in mind, the need for an alternative approach for comparing methods became obvious. A process was needed that not only showed the benefits of using the new algorithm on a computer, but also the intrinsic value of its ease of calculation when done manually (on paper). Crowder, Dembo, and Mulvey suggest the following:

A variety of performance indicators has been traditionally used by mathematical programmers for evaluating the efficiency of competing techniques. A simple counting of the number of steps is required by the algorithm. This indicator is relatively independent of the computer used. Another indicator is the number of times that a basic operation such as addition or multiplication/division is required during the execution of the algorithm [54].

Such a procedure was alluded to by Kronsjö [55] and substantiated by others. This alternative approach bases its criteria on the computational complexity of an algorithm. In using this terminology, Kronsjö refers to estimating the computational effort required to solve the problem and measuring it by the number of arithmetic or logical operations required, e.g., the number of multiplications and additions needed and the number of comparisons between two entries in an iterative algorithm. This approach was also utilized by Winograd in his development of ways to improve the efficiency of computational algorithms. He developed identities relating number of multiplications and additions required to evaluate certain forms of polynomials and later extended this to the total number of arithmetic operations required of an algorithm [55]. This literature and the consultation of Drs. Hooper and Ranganath of the University of Alabama in Huntsville Computer Science Department [56] encouraged the author to develop a procedure termed Computational Complexity Scoring (CCS) system, to compare the performance of the new algorithm against the branch-and-bound method on a given problem. The scoring procedure and comparison methodology will now be discussed in detail in the following section.

B. Algorithm Computational Complexity Comparison

After examining the suggestions of Winograd, Kronsjö, Crowder, et al., and Hooper and Ranganath, a scoring system was developed for the evaluation of these two competing techniques. The scoring system contains three distinct aspects. One is the knowledge of how many specific seed value expansions (explorations) are required to solve the problem. If solving the problem by hand, one simply counts the number of times a point is examined. If the IESIP software is being utilized, it will keep track of this value for the user. The same requirement is true of the branch-and-bound technique. Knowledge of the number of iterations required to solve the problem is needed. An iteration using the branch-and-bound software utilized in this research refers to a conclusive (either feasible or infeasible) iteration of the simplex method obtained from a node investigation.

Another piece of knowledge required by the scoring system is the number of comparisons (e.g., is one value less than another?), additions (subtractions), multiplications, and divisions required of IESIP to do an expansion, or of branch-and-bound to perform a simplex (dual) iteration. Also required is a relative "weight" or duty factor associated with each of these operations. For example, multiplication and division increase computational complexity not only for the person solving the problem by hand, but also for the computer. The latter two aspects, number of operations and relative weights, required further explanation in the following sections.

1. Number of Operations

When an individual performs a branch-and-bound integer programming procedure by hand, he generally is faced with the painstaking ritual of adding, subtracting, multiplying, and dividing numerous sets of fractional numbers at every iteration of this procedure. With IESIP, this task has been reduced to a simple case of adding and subtracting "1," multiplying by integers, and evaluating a constraint (comparison), with no fractional operations or divisions.

Examining IESIP first, and using the following worst case assumptions, the number of additions/subtractions, multiplications, and comparisons can be formulated:

(1) All constraints must be examined each time a point is explored. (Best case would be a constraint failure detection on the first constraint examined.)

(2) Every expansion examines both +1 and -1 increments on each variable. This would be decreased if all coefficients in the objective function were positive. (IESIP rule states that if the +1 increment does not have a constraint failure, then there is no need to examine the -1 increment in a maximization problem.)

Addition/Subtraction Operations:

(1) Incrementation of each variable by +1 or -1 \rightarrow (2) (# of variables)

(2) Addition of each constraint term \rightarrow # variables - 1

(3) Total additions for constraint evaluation \rightarrow (# variables - 1) (# constraints) (2) (# variables).

Therefore, the total number of addition/subtraction operations per IESIP expansion is

$$2v + (v - 1)(c)(2v) = 2v(1 + vc - c) \quad (5)$$

where v is the number of variables and c is the number of constraints.

Multiplication Operations:

(1) Evaluation of the objective function at each increment of 1 \rightarrow (2) (# variables)

(2) Evaluation of each constraint at each increment of 1 \rightarrow (2) (# variables) (# variables) (# constraints).

Therefore, the total number of multiplication operations per IESIP expansion is

$$2v + (2v)(vc) = 2v(1 + vc) \quad (6)$$

Division Operations:

No division is required when using IESIP.

Comparison Operations:

- (1) Checking constraints \rightarrow (2) (# variables)(# constraints)
- (2) Checking the best value against the previous best \rightarrow 1.

Therefore, the total number of comparison operations per IESIP expansion is

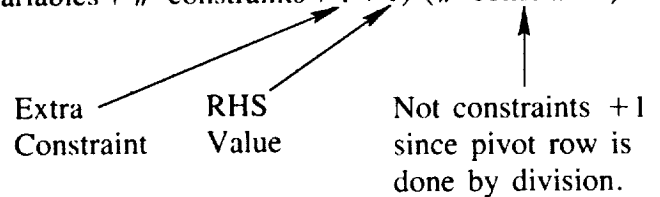
$$2vc + 1 \tag{7}$$

Now we will examine the branch-and-bound procedure. In order to give it the benefit of the doubt, we will make best case assumptions. These are the following:

- (1) Only one additional constraint is added to the simplex problem at each iteration.
- (2) Optimal for a respective iteration is obtained with use of dual simplex only. Sometimes in branch-and-bound, the user must apply regular simplex to find optimal within a particular node branch when dual simplex cannot be continued.

Addition/Subtraction Operations:

- (1) Calculation of each new row \rightarrow (# variables + # constraints + 1 + 1) (# constraints)



- (2) Calculation of x_0 row \rightarrow (# constraints + 1) (# constraints + # variables + 1 + 1).

Therefore, the best case total number of additions/subtractions at each iteration of branch-and-bound is

$$(v + c + 2)(c) + (c + 1)(v + c + 2) = (v + c + 2) (2c + 1) \tag{8}$$

Multiplication Operations: (same as addition/subtraction)

Division Operations:

- (1) Pivot row calculations \rightarrow # variables + # constraints + 1 + 1
- Extra Constraint ↖ RHS
- (2) Ratio row calculations \rightarrow # variables + # constraints + 1.
- ↑
Added Constraint

Therefore, the best case total number of division operations at each iteration of branch-and-bound is

$$v + c + 2 + v + c + 1 = 2v + 2c + 3 \quad (9)$$

Comparison Operations:

- (1) Searching for pivot \rightarrow (# constraints + 1) + (# variables + # constraints + 1)
- (2) Check to see if done \rightarrow (# constraints + 1)
- (3) Compare to best value so far \rightarrow 1.

Therefore, the best case total comparison operations at each iteration of branch-and-bound is

$$c + 1 + v + c + 1 + c + 1 + 1 = v + 3c + 4 \quad (10)$$

Finally, to compute the total number of respective operations for either branch-and-bound or IESIP, one need only multiply the total number of expansions or iterations by the values obtained with the above formulas to obtain the totals for each operation type. An example will follow later in this section that describes the details of this calculation.

2. Weights Associated With Each Operation

It would be difficult to determine the relative difficulty encountered by an individual solving an ILP problem between addition and multiplication calculations. However, a computer examines the two operations on a consistent mechanical field of play and weights the difficulty of each operation according to how much work it has to do to accomplish a respective task. For instance, the Intel 80286 microprocessor-based personal computer can execute a compare instruction in 6 clock cycles, while the same machine requires 25 clock cycles to execute a division instruction. In table 3, the clock period instruction times of a representative sampling of personal computer microprocessors are listed for the four operations required by the two algorithms presented here.

Table 3. Worst case microprocessor instruction times [57,58].

(Values in number of clock periods)

Instruction	Motorola 68010 (1982)	Intel 80386 (1988)	Intel 80286 (1988)	80286 Relative Effort Factors
Compare	6	5	6	1
Add/Sub	12	7	7	1.1666
Multiply	42	24	24	4.0
Divide	122	25	25	4.1666

Note: 80286 Clock Period = 125 nanoseconds/period
 80386 Clock Period = 50 nanoseconds/period

The Intel 80286 is chosen as the benchmark for our comparison for the following two reasons: (1) it is widely available for both student and industry users, and (2) it is the microprocessor that was used by the author to conduct this study. Also shown in the table are the relative weights for each operation. These are calculated by taking the compare operation as the lowest user of computer effort and relating it by a multiplication factor to the other more effort-demanding operations.

The only task remaining to produce a composite computational complexity score for a problem is to multiply the total addition, multiplication, division, and compare operations by their respective effort factors and sum to a total value. A detailed example follows which uses a previously solved problem from section III and the output of the branch-and-bound based ILP software package called QSB.

C. Example Problem

From section III we have:

$$\text{maximize } Z = f(x_1, x_2) = 4,000x_1 + 7,000x_2$$

$$\text{subject to: } 1,200x_1 + 2,000x_2 \leq 6,000$$

$$25,000x_1 + 80,000x_2 \leq 200,000$$

$$x_1, x_2 \geq 0 \text{ and integer.}$$

Using IESIP in section III, we obtained an integer solution of $f(5,0) = 20,000$ after nine expansions (or exploratory searches).

Using QSB, the following branch-and-bound solution is obtained after nine iterations:

Summary of Results for TEST1				Page : 1	
Variables No. Names	Solution	Obj. Fnctn. Coefficient	Variables No. Names	Solution	Obj. Fnctn. Coefficient
1 X1	5.000	4000.000	2 X2	0.000	7000.000
Maximum value of the OBJ = 20000 Total iterations = 9					

Therefore, using the equations from the preceding section we have:

IESIP

$$\text{Additions} \rightarrow 2v(1+vc-c) = 2(2)(1+(2)(2)-2) = 12$$

(from equation (5))

$$\text{Multiplications} \rightarrow 2v(1+vc) = 2(2)(1+(2)(2)) = 20$$

(from equation (6))

$$\text{Divisions} \rightarrow 0$$

$$\text{Comparisons} \rightarrow 2vc+1 = (2)(2)(2)+1 = 9$$

(from equation (7))

Therefore, at nine expansions, the totals are:

108 Addition Operations
 180 Multiplication Operations
 0 Division Operations
 81 Comparison Operations

Now using the effort factors developed, the following computational complexity score can be determined for the IESIP algorithm:

$$CCS_I = 81(1) + 108(1.1666) + 180(4.0) + 0(4.1666) = 927$$

Branch-and-Bound

$$\text{Additions} \rightarrow (v+c+2)(2c+1) = (2+2+2)((2)(2)+1) = 30$$

(from equation (8))

$$\text{Multiplications} \rightarrow (v+c+2)(2c+1) = (2+2+2)((2)(2)+1) = 30$$

(from equation (8))

$$\text{Divisions} \rightarrow 2v+2c+3 = (2)(2)+(2)(2)+3 = 11$$

(from equation (9))

$$\begin{aligned} \text{Comparisons} &\rightarrow v + 3c + 4 = 2 + (3)(2) + 4 = 12 \\ &\text{(from equation (10))} \end{aligned}$$

Therefore, at nine iterations, the totals are

270 Addition Operations
 270 Multiplication Operations
 99 Division Operations
 108 Comparison Operations

Using the effort factors developed, the following computational complexity score can be determined for the branch-and-bound algorithm:

$$CCS_B = 108(1.0) + 270(1.1666) + 270(4.0) + 99(4.1666) = 1,915.5.$$

Since the IESIP algorithm's score (927) is less than the branch-and-bound score (1,915.5), we can say that it was a computational improvement over the branch-and-bound. Furthermore, as an added benefit, it also achieved the same optimal solution as the branch-and-bound procedure— $f(5,0) = 20,000$.

As previously mentioned in section III, the IESIP computer program calculates the exact number of additions, multiplications, and comparisons required by the IESIP algorithm to solve a particular problem. The example just solved by the IESIP software produced the following output:

Best solution(s) found: $(5,0) = 20000.000$

Computation of the rounded optimal solution required:

Additions: 2 Multiplications: 4 Compares: 2

9 Points were expanded requiring

Additions: 87 Multiplications: 174 Compares: 46

Computational complexity score = 863.827.

One can easily ascertain that the IESIP CCS using the software is much lower than the worst case value just computed by hand. This is due to the software's ability to eliminate certain needless computations such as expanding type II searches in both the "minus one" and "plus one" directions after a success of one or the other. In contrast, the manual method presented above assumes worst-case conditions, i.e., every point is expanded with both +1 and -1.

In part D of this section, a table is presented that summarizes the findings obtained using some test problems. The computational complexity score for the IESIP algorithm shown for each test problem is the score found using the software. The score shown for branch-and-bound is the score found by assuming the best case situation described earlier in this section (to give branch-and-bound its best advantage).

Table 4. Computational results of test problems.

Problem # and Type	Number of variables (v) and constraints (c)	B & B Optimal Value *	IESIP Solution Value *	B & B CCS	IESIP CCS	Improvement	Percentage Improvement over B & B	Performance Ratio
1 Max	2v, 2c	20,000 *(9)	20,000 (9)	1915.5	863.8	Yes	54.9 %	1.00
2 Max	2v, 2c	0 (5)	0 (1)	1064.2	88.5	Yes	91.7 %	1.00
3 Max	2v, 2c	Infeasible(2)	Infeasible(1)	425.7	20.3	Yes	95.2 %	1.00
4 Max	2v, 2c	10 (3)	10 (4)	638.5	291.0	Yes	54.4 %	1.00
5 Max	2v, 2c	17 (17)	17 (4)	3618.2	329.7	Yes	90.9 %	1.00
6 Min	2v, 2c	40 (3)	40 (7)	638.5	679.7	No	- 6.5 %	1.00
7 Min	2v, 2c	2 (5)	2 (4)	1064.2	359.2	Yes	66.2 %	1.00
8 Max	2v, 2c	12 (5)	12 (4)	1064.2	340.8	Yes	67.9 %	1.00
9 Max	2v, 2c	4 (11)	4 (4)	2341.2	281.8	Yes	87.8 %	1.00
10 Max	2v, 2c	165 (5)	165 (4)	1064.2	359.2	Yes	66.2 %	1.00
11 Max	2v, 2c	37 (5)	37 (4)	1064.2	359.2	Yes	66.2 %	1.00
12 Max	2v, 2c	11 (5)	11 (4)	1064.2	329.7	Yes	69.0 %	1.00
13 Min	2v, 2c	6 (3)	6 (5)	638.5	397.8	Yes	37.7 %	1.00
14 Max	2v, 2c	13 (5)	13 (5)	1064.2	359.2	Yes	66.2 %	1.00
15 Max	2v, 2c	10 (3)	10 (5)	638.5	388.7	Yes	39.1 %	1.00
16 Max	2v, 2c	57 (5)	57 (5)	1064.2	602.3	Yes	43.4 %	1.00
17 Min	2v, 2c	19 (3)	19 (6)	638.5	504.7	Yes	20.9 %	1.00
18 Max	2v, 2c	5 (7)	5 (9)	1489.3	648.2	Yes	56.5 %	1.00
19 Max	2v, 3c	24 (7)	24 (4)	2256.7	481.2	Yes	78.7 %	1.00
20 Max	2v, 3c	16 (9)	16 (5)	2901.0	677.5	Yes	76.6 %	1.00
21 Max	2v, 3c	102 (3)	102 (5)	967.0	610.3	Yes	36.9 %	1.00
22 Max	3v, 2c	10 (3)	10 (10)	744.0	1919.0	No	-157.9 %	1.00
23 Max	3v, 2c	45 (19)	45 (4)	4712.0	699.7	Yes	85.2 %	1.00
24 Max	3v, 2c	15 (3)	15 (8)	744.0	2090.7	No	-181.0%	1.00
25 Max	3v, 3c	10 (6)	10 (8)	2207.0	1979.0	Yes	10.3 %	1.00

Table 4. Computational results of test problems (continued)

Problem # and Type	Number of variables (v) and constraints (c)	B & B Optimal Value *	IESIP Solution Value *	B & B CCS	IESIP CCS	Improvement	Percentage Improvement over B & B	Performance Ratio
26 Max	3v, 3c	26 (5)	26 (3)	1839.2	824.3	Yes	55.2 %	1.00
27 Max	3v, 3c	47.8 (117)	47.8 (9)	43036.5	2987.3	Yes	93.1 %	1.00
28 Max	3v, 3c	64 (5)	64 (1)	1839.2	362.0	Yes	80.3 %	1.00
29 Max	3v, 3c	23 (5)	23 (3)	1839.2	870.3	Yes	52.7 %	1.00
30 Max	3v, 3c	4 (5)	4 (5)	1839.2	853.0	Yes	53.6 %	1.00
31 Max	3v, 3c	14 (9)	14 (6)	3310.5	1275.3	Yes	61.5 %	1.00
32 Max	3v, 3c	8 (43)	8 (11)	15816.9	2953.6	Yes	81.3 %	1.00
33 Max	3v, 4c	5 (11)	5 (1)	5580.7	454.0	Yes	91.9 %	1.00
34 Max	3v, 4c	27 (13)	27 (1)	6595.3	454.0	Yes	93.1 %	1.00
35 Max	3v, 5c	18 (3)	18 (9)	2008.5	2465.6	No	- 22.8 %	1.00
36 Max	4v, 3c	63 (23)	63 (6)	9506.7	2259.0	Yes	76.2 %	1.00
37 Max	4v, 3c	23 (25)	23 (13)	10333.2	6131.9	Yes	40.7 %	1.00
38 Min	4v, 3c	23 (11)	23 (84)	4546.7	26443.2	No	- 481.6 %	1.00
39 Max	4v, 3c	22 (13)	22 (6)	5373.3	2074.5	Yes	61.4 %	1.00
40 Max	4v, 3c	42 (7)	42 (3)	2893.3	1108.5	Yes	61.7 %	1.00
41 Max	4v, 3c	28 (3)	28 (3)	1240.0	1108.5	Yes	10.6 %	1.00
42 Max	4v, 6c	4250 (49)	4250 (4)	45463.8	2455.5	Yes	94.6 %	1.00
43 Max	6v, 3c	1400 (33)	1400 (9)	16643.0	10320.6	Yes	38.0 %	1.00
44 Max	10v, 4c	90481 (125)	90481 (53)	246812.5	160052.5	Yes	35.2 %	1.00
45 Max	20v, 4c	92564 (207)	92550 (22)	301702.5	279541.2	Yes	7.3 %	1.00015

* Numbers in parenthesis indicate number of iterations and expansions for branch-and-bound and IESIP respectively.

Table 5. Summary of test problem results.

Type of Problem					Improvements Over B & B	No Improvement in CSS over B & B Type	Average % Improvement over B & B for a Class of Problems
# Var.	# Constraints	# Max.	# Min.	# Total			
2	2	14	4	18	17 (94.4 %)	1 (min)	63.2 %
2	3	3	0	3	3 (100 %)	0	64.1 %
3	2	3	0	3	1 (33.3 %)	2 (max)	--
3	3	8	0	8	8 (100 %)	0	61 %
3	4	2	0	2	2 (100 %)	0	92.5 %
3	5	1	0	1	0 (0 %)	1 (max)	--
4	3	5	1	6	5 (83.3 %)	1 (min)	--
4	6	1	0	1	1 (100 %)	0	*
6	3	1	0	1	1 (100 %)	0	*
10	4	1	0	1	1 (100 %)	0	*
20	4	1	0	1	1 (100 %)	0	*
Totals		40	5	45	40	5 (2 min) (3 max)	

Total Percentage of Problems that Showed Improvement over B & B = 88.9 %
 Percentage of Maximization problems that Showed Improvement over B & B = 92.5 %
 Percentage of Minimization problems that Showed Improvement over B & B = 60.0 %
 Average percent improvement in Computational Complexity (all problems) = 35.6 %
 Average percent improvement in Computational Complexity (only improved problems) = 61.2 %

* only one test problem was examined for this class of problem

D. Computational Experience With Test Problems

In order to verify the improved efficiency of the heuristic developed in this research, it is necessary to solve a variety of test problems using the new algorithm. There were 45 problems found in different texts and they vary in size from 2 variables and 2 constraints up to 20 variables and 4 constraints. These test problems and their sources are presented in appendix C. Of the 45 problems, 41 were solved manually as well as with the IESIP software in appendix B. The remaining 4 problems, because of their size, were solved only with the software. Every test problem was also solved using the commercially available branch-and-bound integer programming software entitled QSB. The results of the test problems are presented in tables 4 and 5.

When solved with the IESIP algorithm, 44 of the 45 test problems achieved the same optimum solution as found with the branch-and-bound method. The one remaining problem (20 variables) got a near optimal answer that had a performance ratio (Parker and Rardin optimal/heuristic ratio) of 1.00015. Of the 45 test problems, 40 (89 percent) showed substantial improvement over the branch-and-bound solution examined in light of their respective computational complexity scores.

E. Multiple Optima

Two of the test problems illustrate a major advantage of the IESIP algorithm over the branch-and-bound algorithm. This is its ability to find and identify multiple optimal solutions. As an example, test problem 18 when solved with the branch-and-bound algorithm produces the following results:

Summary of Results for TEST18				Page : 1	
Variables No. Names	Solution	Obj. Fnctn. Coefficient	Variables No. Names	Solution	Obj. Fnctn. Coefficient
1 X1	5.000	1.000	2 X2	0.000	1.000
Maximum value of the OBJ = 5 Total iterations = 7					

Notice that only one solution is produced (5,0) with no indication of multiple optima. However, when the same test problem is solved using IESIP, it produces three optimal solutions in nine expansions—(5,0), (3,2), and (4,1). All three are recognized and printed by the software as shown below:

```
Is the objective function to be maximized?
[y]es/[n]o/[h]elp default = no Y
Do you have the functions in a data file?
[y]es/[n]o/[h]elp default = no Y
Enter the file? TEST18.DAT
```

```
Maximization problem
Number of variables = 2
```

```
Objective function:
+1.00 X1 +1.00 X2
```

```
Constraint function(s):
+2.00 X1 +5.00 X2 <= 16.00
+6.00 X1 +5.00 X2 <= 30.00
```

```
Continuous solution rounded value:
(3, 1)
```


Do you wish to accept this rounded solution?

[y]es/[n]o/[h]elp default = yes Y

Enter the number of initial expansions

Integer or 0 for help

1

Do you wish to see the nodes as they are expanded?

[y]es/[n]o/[h]elp default = no Y

Node Expansion (1)

(3, 1) = 4.000
(4, 1) = 5.000
(2, 1) = 3.000
(3, 2) = 5.000
(3, 0) = 3.000

Node Expansion (5)

(3, 0) = 3.000
(4, 0) = 4.000
(2, 0) = 2.000
(3, 1) = 4.000

Node Expansion (2)

(4, 1) = 5.000
(5, 1) = C.F.
(3, 1) = 4.000
(4, 2) = C.F.
(4, 0) = 4.000

Node Expansion (6)

(3, 1) = 4.000
(4, 1) = 5.000
(3, 2) = 5.000

Press RETURN to continue...

Node Expansion (3)

(3, 2) = 5.000
(4, 2) = C.F.
(2, 2) = 4.000
(3, 3) = C.F.
(3, 1) = 4.000

Node Expansion (7)

(4, 0) = 4.000
(5, 0) = 5.000
(4, 1) = 5.000

Press RETURN to continue...

Node Expansion (4)

(2, 1) = 3.000
(3, 1) = 4.000
(1, 1) = 2.000
(2, 2) = 4.000
(2, 0) = 2.000

Node Expansion (8)

(2, 2) = 4.000
(3, 2) = 5.000
(2, 3) = C.F.
(2, 1) = 3.000

Press RETURN to continue...

Node Expansion (9)

(5, 0) = 5.000
(6, 0) = C.F.
(4, 0) = 4.000
(5, 1) = C.F.

Do you wish to have the problem restated?

[y]es/[n]o/[h]elp default = no Y

Maximization problem

Number of variables = 2

Objective function:

+1.00 X1 +1.00 X2

Constraint function(s):

+2.00 X1 +5.00 X2 <= 16.00

+6.00 X1 +5.00 X2 <= 30.00

Continuous solution rounded value:

(3, 1)

Best solution(s) found:

(5, 0) = 5.000

(3, 2) = 5.000

(4, 1) = 5.000

Computation of the rounded optimal solution required:

Additions: 2 Multiplications: 4 Compares: 2

9 Points were expanded requiring

Additions: 65 Multiplications: 130 Compares: 32

Computational complexity score = 648.162

F. Restrictions Required of IESIP

Associated with many heuristic procedures are certain restrictions placed on the type and size of problem that realistically can be solved. The IESIP algorithm's only restriction is that it cannot accommodate strict "equal to" (=) constraints. This problem is apparent and self-explained when one tries to round a continuous solution to an integer solution when starting the IESIP algorithm.

The following example illustrates the "equal to" constraint dilemma:

maximize $Z = f(x_1, x_2, x_3) = 20x_1 + 10x_2 + 10x_3$

subject to: $2x_1 + 20x_2 + 4x_3 \leq 15$

$6x_1 + 20x_2 + 4x_3 = 20$

$x_1, x_2, x_3 \geq 0$ and integer .

The continuous solution is $f(3.333, 0, 0) = 66.666$. If an attempt is made to apply the IESIP rounding principles, then a suitable starting solution cannot be found. For example, (3,0,0) fails constraint two. So does every other combination except the unique and optimal solution of (2,0,2) = 60. Unless one is "lucky" and the optimal equals the rounded solution, no feasible continuous rounded solution can be found for a problem like this, and therefore the IESIP algorithm has no starting point. Not even (0,0,0) will work.

The only other restriction found for the IESIP algorithm is in its software version. Due to segmentation of memory by MS-DOS, Pascal programs are limited to 640 (655,360) kilobytes of memory. When solving the last test problem (20 variables, four constraints), an attempt was made to extend the type I search to a second level and see if the optimal solution would be found. The program was halted at expansion number 96 due to lack of memory. This problem did not occur with the 10 variable problem that was type I expanded twice. This memory problem will be dependent on the following three factors: (1) number of variables, (2) number of constraints, and (3) number of expansions required to solve the problem. It should be noted that this problem would not occur if the software were simply placed on a mainframe computer with a much larger memory capacity than a typical PC-based system.

V. CONCLUSIONS AND RECOMMENDATIONS

A. Conclusion

The purpose of the search documented in the previous sections was the creation of an improved heuristic search technique for solving pure integer linear programming problems. By drawing from the ideas of Hooke and Jeeves type I and type II exploratory searches, greedy procedures, and neighborhood searches, the IESIP algorithm starts with the continuous solution to the problem which it rounds to a starting integer solution, then expands points until obtaining the best maximum or minimum solution to the problem. Not only does IESIP show significant computational complexity improvements over the branch-and-bound method in 89 percent of the 45 test problems examined, but it also finds the global optimal solution in all but one of these.

In summary, the following five conclusions are drawn from the previous discussion:

1. IESIP has clearly proven itself to be more computationally efficient than the branch-and-bound method in 40 of the 45 test problems that were examined. It also has been shown that IESIP found the same optimal solution as the branch-and-bound method in 44 of the 45 test problems. Furthermore, the one problem that did not achieve the optimal value was only 14 off the optimal which gives a heuristic solution performance ratio of 1.00015.
2. A computational complexity scoring system was developed to compare the number and degree of actual computations required of the IESIP algorithm and the best case number and degree of computations required of the branch-and-bound algorithm. These scores were computed for each test problem and shown in table 4 of section IV. Of the problems that showed improvement, the average percentage improvement in computational effort over the branch-and-bound method was 61.2 percent.
3. IESIP showed ability in solving larger problems. This was illustrated by solving both a 10 and 20 variable problem. It also showed an ability to find far-removed-from-continuous integer solutions (integer solutions that are not close to their rounded continuous solution) with approximately 55 percent less computational effort than branch-and-bound (see problem 1 and problem 15).

4. IESIP was also found to have one other advantage over branch-and-bound in that in two cases (test problem 18 and test problem 25) alternative optima were not only recognized but also identified. The branch-and-bound algorithm only found one of the optimal solutions in each problem and also gave no indication that other solutions exist.

5. Lastly, and probably the most overlooked improvement in integer programming brought about by this research, is the ease of its implementation and execution when one is solving a problem by hand. If a problem (e.g., test problem 1) requires nine iterations of dual simplex to solve by the branch-and-bound algorithm and nine expansions of IESIP, then the computational work involved in manually solving this problem with IESIP is much easier than the nine full tableaus of simplex required if solving it with branch-and-bound. This is primarily due to the IESIP procedure's univariate use of only $+1$ and -1 in its solution process. This improvement of not having to go through the agony of calculating three or four new basic rows with each iteration of simplex is, in the author's opinion, quite valuable, particularly for instructional purposes.

The methodology and heuristic procedure developed and illustrated in the preceding sections represent an improved exploratory search technique for solving certain integer programming problems. The algorithm not only solves two and three variable problems, but also works well on the tougher 6, 10, and 20 variable problems. The only apparent limitation discovered about the algorithm is its inability to deal with "equal to" constraints. However, various combinations of "less than or equal to" and "greater than or equal to" constraints did not affect the algorithm's ability to find an optimal solution. It is this seemingly broad range of applications that will hopefully stimulate future research on the application of this procedure to larger and more complex problems.

B. Recommendations for Future Work

With some minor modifications to the algorithm and software, one could solve nonlinear integer programming (NLIP) problems. The body of knowledge surrounding NLIP problems could be enhanced by possible improvements from a nonlinear IESIP. This topic could be examined from both a nonlinear objective function with linear constraints standpoint and also from a nonlinear objective function and nonlinear constraint situation. IESIP's impact on nonlinear problems will be even better than what was achieved on linear problems, since NLIP's are generally even more cumbersome to work with than linear problems.

Another area of exploration would be the detailed analysis of many more problems of varying variable and constraint size to further substantiate the improved efficiency of IESIP over branch-and-bound. For example, one might determine that if the number of constraints is three times the number of variables, then the likelihood of IESIP having a lower CCS score is greatly reduced. This particular investigation would require large numbers of sample or test problems of various sizes.

Finally, the possibility of extending IESIP to mixed programming problems bears investigation. Certain modifications in the searching procedures designed to separate integer-constrained variables from those without such restrictions (analogous to mixed cutting methods) may be possible.

REFERENCES

1. Nemhauser, G.L., and Wolsey, L.A.: "Integer and Combinatorial Optimization." John Wiley and Sons, New York, 1988, pp. vii and 60.
2. Beale, E.M.: "Integer Programming." Computational Mathematical Programming, Vol. 15, 1985, p. 2.
3. Taha, H.A.: "Integer Programming – Theory, Applications, and Computations." Academic Press, New York, 1975, pp. 2, 6, and 202–225.
4. Garfinkel, R.S., and Nemhauser, G.L.: "Integer Programming." John Wiley and Sons, Inc., New York, 1972, pp. 2–5, 60, 324–337.
5. Taha, H.A.: "Operations Research: An Introduction." MacMillan Publishing Co., New York, 1987, pp. 305–330.
6. Gomory, R.E.: "An Algorithm for Integer Solutions to Linear Programs." Recent Advances in Mathematical Programming, Graves and Wolfe, eds., McGraw-Hill, New York, 1963, pp. 269–302.
7. Gondran, M., and Simeone, B.: "Cutting Planes." Annals of Discrete Mathematics, Vol. 5, 1979, p. 193.
8. Salkin, H.M.: Integer Programming." Addison-Wesley Publishing Co., Reading, Massachusetts, 1975, pp. 17–19.
9. Land, A., and Doig, A.: "An Automatic Method for Solving Discrete Programming Problems." Econometrica, Vol. 28, No. 3, 1960, pp. 497–520.
10. Dakin, R.T.: "A Tree Search for Mixed-Integer Programming Problems." The Computer Journal, Vol. 8, 1965, pp. 250–255.
11. Driebeck, N.: "An Algorithm for the Solution of Mixed Integer Programming Problems." Management Science, Vol. 12, No. 7, 1966, pp. 576–587.
12. Balas, E.: "An Additive Algorithm for Solving Linear Programs With Zero-One Variables." Operations Research, Vol. 13, No. 4, 1965, pp. 517–548.
13. Lemke, C., and Spielberg, K.: "Direct Search Algorithm for Zero-One and Mixed Integer Programming." Operations Research, Vol. 15, No. 5, 1967, pp. 892–914.
14. Geoffrion, A.M.: "An Improved Implicit Enumeration Approach for Integer Programming." Operations Research, Vol. 17, 1969, pp. 437–454.

15. Budnick, F.S., Mojena, R., and Vollmann, T.: "Principles of Operations Research for Management." Richard D. Irwin, Inc., Homewood, Illinois, 1977, p. 229.
16. Ozan, T.M.: "Applied Mathematical Programming for Production and Engineering Management." Prentice-Hall, Englewood Cliffs, New Jersey, 1986, pp. 306–392.
17. Healy, W.C.: "Multiple Choice Programming." *Operations Research*, Vol. 12 1964, pp. 122–138.
18. Reiter, S., and Sherman, G.: "Discrete Optimizing." *Journal of the Society for Industrial and Applied Mathematics*, Vol. 13, No. 3, 1965, pp. 864–889.
19. Reiter, S., and Rice, D.B.: "Discrete Optimizing Solution Procedures for Linear and Non-linear Integer Programming Problems." *Management Science*, Vol. 12, No. 11, 1966, pp. 829–850.
20. Kreuzberger, H.: "Numerische Erfahrungen mit einem heuristischen verfahren zur Lösung ganzzahliger linearer optimierungsprobleme." *Electronische Daten-verarbeitung*, Heft 7, 1970, Seiten 289–306.
21. Kochenberger, G.A., McCarl, B.A., and Wyman, F.P.: "A Heuristic for General Integer Programming." *Decision Sciences*, Vol. 5, 1974, pp. 36–44.
22. Hillier, F.S., and Lieberman, G.J.: "Introduction to Operations Research." Holden-Day, Inc., Oakland, California, 1980, pp. 714, 715, and 740.
23. Cooper, L., and Cooper, M.W.: "All-Integer Linear Programming - A New Approach via Dynamic Programing." *Naval Research Logistics Quarterly*, Vol. 25, No. 1, 1978, pp. 425–429.
24. Marsten, R.E., and Morin, T.L.: "A Hybrid Approach to Discrete Mathematical Programming." *Mathematical Programming*, Vol. 14, 1978, pp. 21–40.
25. Parker, R.G., and Rardin, R.L.: "Discrete Optimization." Academic Press, Inc., Boston, Massachusetts, 1988, pp. 357–383.
26. Cooper, M.W.: "A Survey of Methods for Pure Nonlinear Integer Programming." *Management Science*, Vol. 27, No. 3, 1981, pp. 353–361.
27. Pegden, C.D., and Petersen, C.C.: "An Algorithm for Solving Integer Programming Problems With Separable Nonlinear Objective Functions." *Naval Research Logistics Quarterly*, Vol. 26, 1979, pp. 595–609.
28. Cabot, V.A., and Erenguc, S.S.: "A Branch-and-Bound Algorithm for Solving a Class of Nonlinear Integer Programming Problems." *Naval Research Logistics Quarterly*, Vol. 33, 1986, pp. 559–567.

29. Balas, E.: "Duality in Discrete Programming: The Quadratic Case." *Management Science*, Vol. 16, 1969, pp. 14–32.
30. McBride, R.D., and Yormark, J.S.: "An Implicit Enumeration Algorithm for Quadratic Integer Programming." *Management Science*, Vol. 26, No. 3, 1980, pp. 282–296.
31. Volkovich, O.V., Roshchin, V.A., and Sergienko, I.V.: "Models and Methods of Solution of Quadratic Integer Programming Problems." *Cybernetics*, Vol. 23, No. 3, 1987, pp. 289–305.
32. Wagner, H.M., Giglio, R.J., and Glaser, R.G.: "Preventive Maintenance Scheduling by Mathematical Programming." *Management Science*, Vol. 10, No. 2, 1964, pp. 316–334.
33. Giglio, R.H., and Wagner, H.M.: "Approximate Solutions to the Three-Machine Scheduling Problem." *Operations Research*, Vol. 12, No. 3, 1964, pp. 305–324.
34. Bartholdi, J.J.: "A Guaranteed-Accuracy Round-Off Algorithm for Cyclic Scheduling and Set Covering." *Operations Research*, Vol. 29, No. 3, 1981, pp. 501–511.
35. Baum, S., and Trotter, L.E.: "Finite Checkability for Integer Rounding Properties in Combinatorial Programming Problems." *Mathematical Programming*, Vol. 22, 1982, pp. 141–147.
36. Boffey, T.B., and Green, J.R.: "Design of Electricity Supply Networks." *Discrete Applied Mathematics*, Vol. 5, 1983, pp. 25–38.
37. Dillon, T.S., and Edwin, K.W.: "Integer Programming Approach to the Problem of Optimal Unit Commitment With Probabilistic Reserve Determination." *IEEE Transactions on Power Apparatus and Systems*, Vol. 97, No. 6, 1978, pp. 2154–2164.
38. Garver, L.L.: "Transmission Network Estimation Using Linear Programming." *IEEE Transactions on Power Apparatus and Systems*, Vol. 89, No. 7, 1970, pp. 1688–1696.
39. Kaltenbach, J.-C., Preschon, J., and Gehrig, E.H.: "A Mathematical Optimization Technique for the Expansion of Electric Power Transmission Systems." *IEEE Transactions on Power Apparatus and Systems*, Vol. 89, No. 1, 1970, pp. 113–119.
40. Tryfos, P.: "An Integer Programming Approach to the Apparel Sizing Problem." *Journal of the Operational Research Society*, Vol. 37, No. 10, 1986, pp. 1001–1006.
41. Cooper, M.W., and Farhangian, K.: "An Integer Programming Algorithm for Portfolio Selection With Fixed Charges." *Naval Research Logistics Quarterly*, Vol. 29, No. 1, 1982, pp. 147–150.
42. Mehta, R.P.: "Optimizing Returns With Stock Option Strategies." *Computers and Operations Research*, Vol. 9, No. 3, 1982, pp. 233–242.
43. Smith, B.M.: "IMPACS – A Bus Crew Scheduling System Using Integer Programming." *Mathematical Programming*, Vol. 42, 1988, pp. 181–187.

44. Dopazo, J.F., and Merrill, H.M.: "Optimal Generator Maintenance Scheduling Using Integer Programming." *IEEE Transactions on Power Apparatus and Systems*, Vol. 94, No. 5, 1975, pp. 1537-1544.
45. Hooke, R., and Jeeves, T.A.: "A 'Direct Search' Solution of Numerical and Statistical Problems." *Journal of the Association of Computing Machines*, Vol. 8, 1961, pp. 212-229.
46. Bazaraa, M., and Shetty, C.M.: "Nonlinear Programming." John Wiley and Sons, New York, 1979, p. 275.
47. Gottfried, B.S., and Weisman, J.: "Introduction to Optimization Theory." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973, p. 117.
48. Foulds, L.R.: "Optimization Techniques: An Introduction." Springer-Verlag, New York, 1981, p. 335.
49. Wismer, D.A., and Chattergy, R.: "Introduction to Nonlinear Optimization." North-Holland, New York, 1978, p. 272.
50. Claycombe, W.W., and Sullivan, W.G.: "Foundation of Mathematical Programming." Reston Publishing Co., Inc., Reston, Virginia, 1975, pp. 194-199.
51. Ignizio, J.P.: "On the Establishment of Standards for Comparing Algorithm Performance." *TIMS Interfaces*, Vol. 2, No. 1, 1971, pp. 8-11.
52. McCabe, T.J.: "A Complexity Measure." *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, 1976, pp. 308-320.
53. Lee, C.C., and Lee, D.T.: "A Simple On-Line Bin Packing Algorithm." *Journal of the Association of Computing Machinery*, Vol. 32, No. 3, 1985, pp. 562-572.
54. Crowder, H.P., Dembo, R.S., and Mulvey, J.M.: "Reporting Computational Experiments in Mathematical Programming." *Mathematical Programming*, Vol. 15, 1978, pp. 316-329.
55. Kronsjö, L.J.: "Algorithms: Their Complexity and Efficiency." John Wiley and Sons, New York, 1979, pp. 2-5, 131-136.
56. Hooper, J.W., and Ranganath, H.: University of Alabama in Huntsville Computer Science Department, verbal consultation with author and John N. Lovett, Jr., February 22, 1990.
57. Intel Corporation: "Microprocessor and Peripheral Handbook - Vol. 1," 1988, pp. 3.45-3.54, 4.109-4.122.
58. Motorola, Inc.: MC68010 16-Bit Virtual Memory Microprocessor, 1982, pp. 7-8.

BIBLIOGRAPHY

- Adby, P.R., and Dempster, M.A.: "Introduction to Optimization Methods." John Wiley and Sons, New York, New York, 1974.
- Avriel, M.: "Nonlinear Programming." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.
- Balas, E.: "An Additive Algorithm for Solving Linear Programs With Zero-One Variables." *Operations Research*, Vol. 13, No. 4, 1965, pp. 517-548.
- Balas, E.: "Duality in Discrete Programming: The Quadratic Case." *Management Science*, Vol. 16, 1969, pp. 14-32.
- Balas, E., and Guignard, M.: "Branch and Bound/Implicit Enumeration." *Annals of Discrete Mathematics*, Vol. 5, 1979, pp. 185-191.
- Balas, E., and Ho, A.: "Set Covering Algorithms Using Cutting Planes, Heuristics, and Subgradient Optimization: A Computational Study." *Mathematical Programming*, Vol. 12, 1980, pp. 37-60.
- Balinski, M.L.: "Integer Programming: Methods, Uses, and Computation." *Management Science*, Vol. 12, No. 3, 1965, pp. 253-315.
- Bartholdi, J.J.: "A Guaranteed-Accuracy Round-Off Algorithm for Cyclic Scheduling and Set Covering." *Operations Research*, Vol. 29, No. 3, 1981, pp. 501-511.
- Baum, S., and Trotter, L.E.: "Finite Checkability for Integer Rounding Properties in Combinatorial Programming Problems." *Mathematical Programming*, Vol. 22, 1982, pp. 141-147.
- Bazaraa, M., and Shetty, C.M.: "Nonlinear Programming." John Wiley and Sons, New York, New York, 1979.
- Beale, E.M.: "Branch-and-Bound Methods for Mathematical Programming Systems." *Annals of Discrete Mathematics*, Vol. 5, 1979, pp. 201-219.
- Beale, E.M.: "Integer Programming." *Computational Mathematical Programming*, Vol. 15, 1985, pp. 1-53.
- Beale, E.M.: "The Evolution of Mathematical Programming Systems." *Journal of Operations Research Society*, Vol. 36, No. 5, 1985, pp. 357-366.
- Benichou, M., Gauthier, J.M., Hentges, G., and Ribiere, G.: "The Efficient Solution of Large-Scale Linear Programming Problems - Some Algorithms, Techniques, and Computational Results." *Mathematical Programming*, Vol. 13, 1977, pp. 280-322.
- Boffey, T.B., and Green, J.R.: "Design of Electricity Supply Networks." *Discrete Applied Mathematics*, Vol. 5, 1983, pp. 25-38.
- Bradley, G.H., Wahi, P.N.: "An Algorithm for Integer Linear Programming: A Combined Algebraic and Enumeration Approach." *Operations Research*, Vol. 21, No. 1, 1973, pp. 45-60.
- Budnick, F.S., Mojena, R., and Vollmann, T.: "Principles of Operations Research for Management." Richard D. Irwin, Inc., Homewood, Illinois, 1977.
- Bunday, B.D.: "BASIC Optimization Methods." Edward Arnold Publishing, Baltimore, Maryland, 1984.
- Bunday, B.D., and Garside, G.R.: "Optimization Methods in PASCAL." Edward Arnold Publishing, Baltimore, Maryland, 1987.
- Cabot, A.V., and Erenguc, S.S.: "A Branch and Bound Algorithm for Solving a Class of Nonlinear Integer Programming Problems." *Naval Research Logistics Quarterly*, Vol. 33, 1986, pp. 559-567.

- Claycombe, W.W., and Sullivan, W.G.: "Foundations of Mathematical Programming." Reston Publishing Co., Inc., Reston, Virginia, 1975.
- Conley, W.: "Computer Optimization Techniques." Petrocelli Books, Inc., New York, New York, 1984.
- Cooper, L., and Cooper, M.W.: "All-Integer Linear Programming - A New Approach Via Dynamic Programming." *Naval Research Logistics Quarterly*, Vol. 25, No. 1, 1978, pp. 415-429.
- Cooper, M.W.: "A Survey of Methods for Pure Nonlinear Integer Programming." *Management Science*, Vol. 27, No. 3, 1981, pp. 353-361.
- Cooper, M.W.: "Nonlinear Integer Programming for Various Forms of Constraints." *Naval Research Logistics Quarterly*, Vol. 29, No. 4, 1982, pp. 585-592.
- Cooper, M.W., and Farhangian, K.: "An Integer Programming Algorithm for Portfolio Selection With Fixed Charges." *Naval Research Logistics Quarterly*, Vol. 29, No. 1, 1982, pp. 147-150.
- Crowder, H.P., Dembo, R.S., and Mulvey, J.M.: "Reporting Computational Experiments in Mathematical Programming." *Mathematical Programming*, Vol. 15, 1978, pp. 316-329.
- Dakin, R.T.: "A Tree Search for Mixed Integer Programming Problems." *The Computer Journal*, Vol. 8, 1965, pp. 250-255.
- Dillon, T.S., and Edwin, K.W.: "Integer Programming Approach to the Problem of Optimal Unit Commitment With Probabilistic Reserve Determination." *IEEE Transactions on Power Apparatus and Systems*, Vol. 97, No. 6, 1978, pp. 2154-2164.
- Driebeck, N.: "An Algorithm for the Solution of Mixed Integer Programming Problems." *Management Science*, Vol. 12, No. 7, 1966, pp. 576-587.
- Dopazo, J.F., and Merrill, H.M.: "Optimal Generator Maintenance Scheduling Using Integer Programming." *IEEE Transactions on Power Apparatus and Systems*, Vol. 94, No. 5, 1975, pp. 1537-1544.
- Echols, R.E., and Cooper, L.: "Solution of Integer Linear Programming Problems by Direct Search." *Journal of the Association for Computing Machinery*, Vol. 15, No. 1, 1968, pp. 75-84.
- Ecker, J.G., and Kupferschmid, M.: "Introduction to Operations Research." John Wiley and Sons, New York, New York, 1988.
- Edler, J., Nikiforuk, P.N., and Tinker, E.B.: "A Comparison of the Performance Techniques for Direct, On-Line Optimization." *The Canadian Journal of Chemical Engineering*, Vol. 48, 1970, pp. 432-440.
- Evans, J.R.: "Structural Analysis of Local Search Heuristics in Combinatorial Optimization." *Computers and Operations Research*, Vol. 14, No. 6, 1987, pp. 465-477.
- Faaland, B.H., and Hillier, F.S.: "Interior Methods for Heuristic Integer Programming Procedures." *Operations Research*, Vol. 27, No. 6, 1979, pp. 1069-1087.
- Fogiel, M.: "The Operations Research Problem Solver." Research and Education Association, New York, New York, 1987.
- Foulds, L.R.: "Optimization Techniques: An Introduction." Springer-Verlag, New York, New York, 1981.
- Gabbani, D., and Magazine, M.: "An Iterative Heuristic Approach for Multi-Objective Integer-Programming Problems." *Journal of the Operations Research Society*, Vol. 37, No. 3, 1986, pp. 285-291.
- Garfinkel, R.S., and Nemhauser, G.L.: "Integer Programming." John Wiley and Sons, Inc., New York, New York, 1972.
- Garver, L.L.: "Transmission Network Estimation Using Linear Programming." *IEEE Transactions on Power Apparatus and Systems*, Vol. 89, No. 7, 1970, pp. 1688-1696.

- Geoffrion, A.M.: "An Improved Implicit Enumeration Approach for Integer Programming." *Operations Research*, Vol. 17, 1969, pp. 437–454.
- Geoffrion, A.M., and Marsten, R.E.: "Integer Programming Algorithms: A Framework and State-of-the-Art Survey." *Management Science*, Vol. 18, No. 9, 1972, pp. 465–491.
- Giglio, R.J., and Wagner, H.M.: "Approximate Solutions to the Three-Machine Scheduling Problem." *Operations Research*, Vol. 12, No. 3, 1964, pp. 305–324.
- Glover, F.: "A Multi-Phase Dual Algorithm for the Zero-One Integer Programming Problem." *Operations Research*, Vol. 13, No. 6, 1965, pp. 879–929.
- Glover, F.: "Improved Linear Integer Programming Formulations of Nonlinear Integer Problems." *Management Science*, Vol. 22, No. 4, 1975, pp. 455–460.
- Glover, F.: "Reducing the Size of Some IP Formulations by Substitution." *Operational Research Quarterly*, Vol. 27, No. 1, 1976, pp. 261–263.
- Gomory, R.E.: "An Algorithm for Integer Solutions to Linear Programs." *Recent Advances in Mathematical Programming*, Graves and Wolfe, eds., McGraw-Hill, New York, New York, 1963.
- Gondran, M., and Simeone, B.: "Cutting Planes." *Annals of Discrete Mathematics*, Vol. 5, 1979, pp. 193–194.
- Gottfried, B.S., and Weisman, J.: "Introduction to Optimization Theory." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.
- Greenberg, H.J.: "Design and Implementation of Optimization Software." Sijthoff and Noordhoff, The Netherlands, 1978.
- Gupta, O.K., and Ravindran, A.: "Branch and Bound Experiments in Convex Nonlinear Integer Programming." *Management Science*, Vol. 31, No. 12, 1985, pp. 1533–1546.
- Healy, W.C.: "Multiple Choice Programming." *Operations Research*, Vol. 12, 1964, pp. 122–138.
- Hillier, F.S.: "Efficient Heuristic Procedures for Integer Linear Programming With an Interior." *Operations Research*, Vol. 17, 1969, pp. 600–637.
- Hillier, F.S., and Lieberman, G.J.: "Introduction to Operations Research." Holden-Day, Inc., Oakland, California, 1980.
- Himmelblau: "Applied Nonlinear Programming." McGraw-Hill Book Co., New York, New York, 1972.
- Hooke, R., and Jeeves, T.A.: "A 'Direct Search' Solution of Numerical and Statistical Problems." *Journal of the Association of Computing Machines*, Vol. 8, 1961, pp. 212–229.
- Hooper, J.W., and Ranganath, H.: University of Alabama in Huntsville Computer Science Department, verbal consultation with author and J.N. Lovett, Jr., February 22, 1990.
- Hu, T.C.: "Integer Programming and Network Flows." Addison-Wesley Publishing Co., Reading, Massachusetts, 1970.
- Ignizio, J.P.: "On the Establishment of Standards for Comparing Algorithm Performance." *TIMS Interfaces*, Vol. 2, No. 1, 1971, pp. 8–11.
- Intel Corporation: "Microprocessor and Peripheral Handbook, Vol. 1." Intel Corp., Santa Clara, California, 1988.
- Jacoby, S.L., Kawalik, J.S., and Pizzo, J.R.: "Iterative Methods for Nonlinear Optimization Problems." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.
- Jeroslow, R.G.: "There Cannot Be Any Algorithm for Integer Programming with Quadratic Constraints." *Operations Research*, Vol. 21, No. 1, 1973, pp. 221–224.
- Kabe, D.G.: "On Solving Integer Programming Problems." *The Journal of Industrial Mathematics Society*, Vol. 32, No. 2, 1982, pp. 103–123.

- Kaltenbach, J.-C., Peschon, J., and Gehrig, E.H.: "A Mathematical Optimization Technique for the Expansion of Electric Power Transmission Systems." *IEEE Transactions on Power Apparatus and Systems*, Vol. 89, No. 1, 1970, pp. 113-119.
- Kennedy, D.: "Some Branch and Bound Techniques for Nonlinear Optimization." *Mathematical Programming*, Vol. 42, 1988, pp. 147-157.
- Kochenberger, G.A., McCarl, B.A., and Wyman, F.P.: "A Heuristic for General Integer Programming." *Decision Sciences*, Vol. 5, 1974, pp. 36-44.
- Körner, F.: "A New Branching Rule for the Branch and Bound Algorithm for Solving Non-linear Integer Programming Problems." *BIT*, Vol. 28, No. 3, 1988, pp. 701-708.
- Kreuzberger, H.: "Numerische Erfahrungen mit einem heuristischen verfahren zur Lösung ganzzahliger linearer optimierungsprobleme." *Electronische Daten-verarbeitung*, Heft 7/1970, Seiten 289-306.
- Kronsjö, L.I.: "Algorithms: Their Complexity and Efficiency." John Wiley and Sons, New York, New York, 1979.
- Kunzi, H.P., Tzschach, H.G., and Zehnder, C.A.: "Numerical Methods of Mathematical Optimization." Academic Press, New York, New York, 1971.
- Land, A., and Doig, A.: "An Automatic Method for Solving Discrete Programming Problems." *Econometrica*, Vol. 28, No. 3, 1960, pp. 497-520.
- Lau, H.T.: "Combinatorial Heuristic Algorithms With FORTRAN." Springer-Verlag, New York, New York, 1986.
- Lawler, E.L., and Bell, M.D.: "A Method for Solving Discrete Optimization Problems." *Operations Research*, Vol. 14, 1966, pp. 1098-1112.
- Lawrence, J.P., and Steiglitz, K.: "Randomized Pattern Search." *IEEE Transactions on Computers*, April 1972, pp. 382-385.
- Lee, C.C., and Lee, D.T.: "A Simple On-Line Bin Packing Algorithm." *Journal of the Association of Computing Machinery*, Vol. 32, No. 3, 1985, pp. 502-572.
- Lemke, C., and Spielberg, K.: "Direct Search Algorithm for Zero-One and Mixed Integer Programming." *Operations Research*, Vol. 15, No. 5, 1967, pp. 892-914.
- Lev, B., and Weiss, H.J.: "Introduction to Mathematical Programming." North Holland, New York, New York, 1982.
- Lin, B.W., and Rardin, R.L.: "Controlled Experimental Design for Statistical Comparison of Integer Programming Algorithms." *Management Science*, Vol. 25, No. 12, 1980, pp. 1258-1271.
- Little, J.K., Murty, D.S., and Karel, C.: "An Algorithm for the Traveling Salesman Problem." *Operations Research*, Vol. 11, No. 5, 1963, pp. 892-914.
- Llewellyn, R.W.: "Linear Programming." Holt, Rinehart, and Winston, Inc., New York, New York, 1964.
- Markowitz, H.M., and Manne, A.S.: "On the Solution of Discrete Programming Problems." *Econometrica*, Vol. 25, No. 1, 1957, pp. 84-110.
- Marsten, R.E., and Morin, T.L.: "A Hybrid Approach to Discrete Mathematical Programming." *Mathematical Programming*, Vol. 14, 1978, pp. 21-40.
- McBride, R.D., and Yormark, J.S.: "An Implicit Enumeration Algorithm for Quadratic Integer Programming." *Management Science*, Vol. 26, No. 3, 1980, pp. 282-296.
- McCabe, T.J.: "A Complexity Measure." *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, 1976, pp. 308-320.
- McMillan, C.: "Mathematical Programming." John Wiley and Sons, Inc., New York, New York, 1975.

- Mehta, R.P.: "Optimizing Returns With Stock Option Strategies." *Computers and Operations Research*, Vol. 9, No. 3, 1982, pp. 233-242.
- Motorola, Inc.: "MC68010 16-Bit Virtual Memory Microprocessor." Motorola, Inc., Austin, Texas, 1982.
- Nauss, R.M.: "On the Use of Internal Rate of Return in Linear and Integer Programming." *Operations Research Letters*, Vol. 7, No. 6, 1988, pp. 285-289.
- Nemhauser, G.L., and Wolsey, L.A.: "Integer and Combinatorial Optimization." John Wiley and Sons, Inc., New York, New York, 1988.
- Onyekwelu, D.C.: "Computational Viability of a Constraint Aggregation Scheme for Integer Linear Programming Problems." *Operations Research*, Vol. 31, No. 4, 1983, pp. 795-801.
- Ozan, T.M.: "Applied Mathematical Programming for Production and Engineering Management." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.
- Papadimitriou, C.H., and Steiglitz, K.: "Combinatorial Optimization." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.
- Parker, R.G., and Rardin, R.L.: "Discrete Optimization." Academic Press, Inc., New York, New York, 1988.
- Pearl, J.: "Knowledge Versus Search: A Quantitative Analysis Using A*." *Artificial Intelligence*, Vol. 20, 1983, pp. 1-13.
- Pegden, C.D., and Petersen, C.C.: "An Algorithm for Solving Integer Programming Problems With Separable Nonlinear Objective Functions." *Naval Research Logistics Quarterly*, Vol. 26, 1979, pp. 595-609.
- Rao, S.S.: "Optimization: Theory and Applications." John Wiley and Sons, New York, New York, 1984.
- Reiter, S., and Sherman, G.: "Discrete Optimizing." *Journal of the Society for Industrial and Applied Mathematics*, Vol. 13, No. 3, 1965, pp. 864-889.
- Reiter, S., and Rice, D.B.: "Discrete Optimizing Solution Procedures for Linear and Nonlinear Integer Programming Problems." *Management Science*, Vol. 12, No. 11, 1966, pp. 829-850.
- Roth, R.H.: "An Approach to Solving Linear Discrete Optimization Problems." *Journal of the Association for Computing Machinery*, Vol. 17, No. 2, 1970, pp. 303-313.
- Salkin, H.M.: "Integer Programming." Addison-Wesley Publishing Co., Reading, Massachusetts, 1975.
- Sharma, J., and Venkateswaran, K.V.: "A Direct Method for Maximizing the System Reliability." *IEEE Transactions on Reliability*, Vol. R-20, No. 4, 1971, pp. 256-259.
- Sherali, H.D., Staschus, K., and Haucuz, J.M.: "An Integer Programming Approach and Implementation for an Electric Utility Capacity Planning Problem With Renewable Energy Sources." *Management Science*, Vol. 33, No. 7, 1987, pp. 831-847.
- Smith, B.M.: "IMPACS - A Bus Crew Scheduling System Using Integer Programming." *Mathematical Programming*, Vol. 42, 1988, pp. 181-187.
- Song, H.: "Optimum Decision Tree Development Using Entropy." Ph.D. Dissertation, University of Alabama in Huntsville, 1989.
- Spielberg, K.: "Enumerative Methods in Integer Programming." *Annals of Discrete Mathematics*, Vol. 5, 1979, pp. 139-183.
- Sweeney, D.J., and Murphy, R.A.: "A Method of Decomposition for Integer Programs." *Operations Research*, Vol. 27, No. 6, 1979, pp. 1128-1141.
- Syslo, M.M., Deo, N., and Kowalik, J.S.: "Discrete Optimization Algorithms." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.

Taha, H.A.: "Operations Research: An Introduction." MacMillan Publishing Co., New York, New York, 1987.

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

Tillman, F.A., and Littschwager, J.M.: "Integer Programming Formulation of Constrained Reliability Problems." *Management Science*, Vol. 13, No. 11, 1967, pp. 887–899.

Tomlin, J.A.: "An Improved Branch-and-Bound Method for Integer Programming." *Operations Research*, Vol. 19, No. 4, 1971, pp. 1070–1075.

Trauth, C.A., and Woolsey, R.E.: "Integer Linear Programming: A Study in Computational Efficiency." *Management Science*, Vol. 15, No. 9, 1969, pp. 481–493.

Tryfos, P.: "An Integer Programming Approach to the Apparel Sizing Problem." *Journal of the Operational Research Society*, Vol. 37, No. 10, 1986, pp. 1001–1006.

Volkovich, O.V., Roshchin, V.A., and Sergienko, I.V.: "Models and Methods of Solution of Quadratic Integer Programming Problems." *Cybernetics*, Vol. 23, No. 3, 1987, pp. 289–305.

Wagner, H.M., Giglio, R.J., and Glaser, R.G.: "Preventive Maintenance Scheduling by Mathematical Programming." *Management Science*, Vol. 10, No. 2, 1964, pp. 316–334.

Wismer, D.A., and Chattergy, R.: "Introduction to Nonlinear Optimization." North-Holland, New York, New York, 1978.

Wood, C.F.: "Application of 'Direct Search' to the Solution of Engineering Problems." Westinghouse Resources Laboratory Science Paper, 6-41210-1-P1, 1960.

Young, R.D.: "A Simplified Primal (All-Integer) Integer Programming Algorithm." *Operations Research*, Vol. 16, 1968, pp. 750–782.

Zionts, S.: "Linear and Integer Programming." Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974.

APPENDIX A
FRACTIONAL CUT AND BRANCH-AND-BOUND EXAMPLES

Fractional Cut Algorithm

The first requirement of using a fractional cut algorithm is that all the coefficients and the RHS constant of each one of the constraints must be an integer. For example, the constraint

$$1/2x_1 + 2/3x_2 \leq 7/3$$

must be transformed into $3x_1 + 4x_2 \leq 14$ so that no fractions are present. This is imposed due to the algorithm requirement that both regular and slack variables must be integer values.

The next step is to solve the problem as a regular linear programming problem. Once this is done, then we have a final optimal tableau for the linear program in the following form:

Basis	x_1	x_i	x_m	w_1	w_j	w_n	RHS
x_1	1	0	0	α_1^1	...	α_1^j	...	α_1^n	β_1
.
.
x_i	0		1		0	α_i^1		α_i^j		α_i^n	β_i
.
.
x_m	0	0	1	α_m^1	...	α_m^j	...	α_m^n	β_m
z_j	0	0	0	c_1	c_j	c_n	β_0

The variables x_i ($i=1,2,\dots,m$) represent the basic variables and the variables w_j ($j=1,2,\dots,n$) represent the non-basic variables of the optimal solution. β_i ($i=1,2,\dots,m$) are the right hand side values (solutions) to the

corresponding variables in the "Basis" column. β_0 is the present value of the objective function. The Z_j row contains the sensitivity coefficients for each of the variables. The values α_i^j ($i=1,2,\dots,m$ and $j=1,2,\dots,n$) are the slack variable coefficients of the constraints.

$$\text{Let } \beta_i = [\beta_i] + f_i, \quad \alpha_i^j = [\alpha_i^j] + f_{ij}$$

where $N = [a]$ is the largest integer such that $N \leq a$. Therefore, $0 < f_i < 1$ and $0 \leq f_{ij} < 1$. In other words, f_i is a positive fraction and f_{ij} is a nonnegative fraction.

Consider the example

$$\begin{array}{lll} a = 3/2 & [a] = 1 & f_i = 1/2 \\ a = -7/3 & [a] = -3 & f_i = 2/3 \end{array}$$

The source row will be

$$f_i - \sum_{j=1}^n f_{ij}w_j = x_i - [\beta_i] + \sum_{j=1}^n [\alpha_i^j]w_j$$

Since x_i , $[\beta_i]$, $[\alpha_i^j]$ and w_j are integers, the RHS must also be integer as well as the LHS,

$$\begin{array}{l} \text{or} \quad f_i - \sum_{j=1}^n f_{ij}w_j \leq 0 \\ \sum_{j=1}^n f_{ij}w_j - f_i \geq 0 \end{array}$$

We have a CUT_i as a nonnegative slack variable which by definition must be an integer. This constraint equation is the fractional cut.

$$CUT_i = \sum_{j=1}^n f_{ij}w_j - f_i \quad (A1)$$

The algorithm is called a fractional method because all the nonzero coefficients of the generated cut are less than 1.

Two major drawbacks to the fractional method are

1. Roundoff errors may yield incorrect optimal solutions.
2. All solutions are infeasible (noninteger) until the optimal is reached.

The above fractional cut equation can be utilized for every row in the tableau. The strongest cut or largest f_i value is generally used. This is determined by operating rules such as the choice of the CUT_i having

$$\max(f_i) \quad \text{or} \quad \max\left(\frac{f_i}{\sum f_{ij}}\right)$$

Lastly, we apply the dual simplex method to obtain an optimal solution.

The following stopping rule for Fractional Cut is employed: If the new solution is an integer one, stop. If not, construct new CUT_i for the remaining rows and apply the best one to a new row and repeat the algorithm. The following example from Ozan [16] illustrates the fractional cut procedure:

$$\text{Maximize } Z = f(x_1, x_2, x_3) = 2x_1 + x_2 + 2x_3$$

$$\text{Subject to } \begin{array}{l} 2x_1 + x_2 + x_3 \leq 9 \\ x_1 + 2x_2 + 3x_3 \leq 8 \end{array}$$

$$x_1, x_2, x_3 \geq 0 \text{ and integer}$$

Step 1. Solve for optimal (noninteger) solution.

I.

		2	1	2	0	0	0
	Basis	x_1	x_2	x_3	s_1	s_2	Sol
0	s_1	2	1	1	1	0	9
0	s_2	1	2	3	0	1	8
	Z	-2	-1	-2	0	0	0

II.

		2	1	2	0	0	0
	Basis	x_1	x_2	x_3	s_1	s_2	Sol
0	s_1	5/3	1/3	0	1	-1/3	19/3
2	x_3	1/3	2/3	1	0	1/3	8/3
	Z	-4/3	1/3	0	0	2/3	16/3

III.

		2	1	2	0	0	0
	Basis	x_1	x_2	x_3	s_1	s_2	Sol
2	x_1	1	1/5	0	3/5	-1/5	19/5
2	x_3	0	3/5	1	-1/5	2/5	7/5
	Z	0	3/5	0	4/5	2/5	52/5

Therefore, the optimal non-integer solution is $x_1=19/5$,
 $x_2=0$, $x_3=7/5$, $Z=52/5$.

Now we apply the cutting procedure.

$$\begin{aligned}
 x_1 \text{ row} \implies \beta_1 &= 19/5 = 3 + 4/5 \quad (f_1 = 4/5) \\
 \alpha_1^1 &= 1/5 = 0 + 1/5 \quad (f_{11} = 1/5) \\
 \alpha_1^2 &= 3/5 = 0 + 3/5 \quad (f_{12} = 3/5) \\
 \alpha_1^3 &= -1/5 = -1 + 4/5 \quad (f_{13} = 4/5)
 \end{aligned}$$

Therefore, $CUT_1 = 1/5x_2 + 3/5s_1 + 4/5s_2 - 4/5 \implies CUT_1 -$

$$1/5x_2 - 3/5s_1 - 4/5s_2 = -4/5$$

$$x_3 \text{ row} \Rightarrow \beta_1 = 7/5 = 1 + 2/5 \quad (f_2 = 2/5)$$

$$\alpha_2^1 = 3/5 = 0 + 3/5 \quad (f_{21} = 3/5)$$

$$\alpha_2^2 = -1/5 = -1 + 4/5 \quad (f_{22} = 4/5)$$

$$\alpha_2^3 = 2/5 = 0 + 2/5 \quad (f_{23} = 2/5)$$

$$\text{Therefore, } \text{CUT}_2 = 3/5x_2 + 4/5s_1 + 2/5s_2 - 2/5 \Rightarrow \text{CUT}_2 -$$

$$3/5x_2 - 4/5s_1 - 2/5s_2 = -2/5$$

$\text{Max}(f_i) = \max(4/5, 2/5) = 4/5$. Therefore, use x_1 row cut values and apply Dual Simplex.

Therefore, for CUT_1 , we have

	2	1	2	0	0	0		
Basis	x_1	x_2	x_3	s_1	s_2	CUT_1	Sol	
2	x_1	1	1/5	0	3/5	-1/5	0	19/5
2	x_3	0	3/5	1	-1/5	2/5	0	7/5
	CUT_1	0	-1/5	0	-3/5	-4/5	1	-4/5
	Z	0	3/5	0	4/5	2/5	0	52/5
Ratio	-	$\frac{3/5}{-1/5}$ -3	-	$\frac{4/5}{-3/5}$ -4/3	$\frac{2/5}{-4/5}$ -1/2	-		

	2	1	2	0	0	0		
Basis	x_1	x_2	x_3	s_1	s_2	CUT_1	Sol	
2	x_1	1	1/4	0	3/4	0	-1/4	4
2	x_3	0	1/2	1	-1/2	0	1/2	1
0	s_2	0	1/4	0	3/4	1	-5/4	1
	Z	0	1/2	0	1/2	0	1/2	10

All positive integers are obtained, so the optimal solution is reached. Hence, the solution in integer values for the problem is $x_1 = 4$, $x_2 = 0$, $x_3 = 1$ and $Z = 10$.

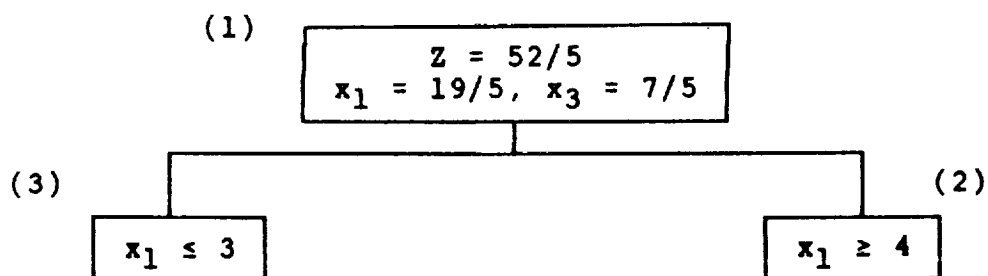
Branch and Bound Example

Using the same continuous optimal solution we obtained from our Fractional Cut example, we have

$$\begin{aligned} \text{Maximize } Z &= f(x_1, x_2, x_3) = 2x_1 + x_2 + 2x_3 \\ \text{Subject to } 2x_1 + x_2 + x_3 &\leq 9 \\ x_1 + 2x_2 + 3x_3 &\leq 8 \\ x_1, x_2, x_3 &\geq 0 \text{ and integer} \\ x_1 = 19/5, x_2 = 0, x_3 = 7/5, Z &= 52/5 \end{aligned}$$

Since x_1 and x_3 are fractional, either can be selected to generate subproblems.

Let us use $x_1 \Rightarrow$ either $x_1 \leq 3$ or $x_1 \geq 4$.



(2) Put $x_1 \geq 4$ into the optimal continuous tableau $\Rightarrow -x_1 \leq 4$.

	Basis	x_1	x_2	x_3	s_1	s_2	s_3	Sol
2	x_1	1	1/5	0	3/5	-1/5	0	19/5
2	x_3	0	3/5	1	-1/5	2/5	0	7/5
0	s_3	-1	0	0	0	0	1	-4
	Z	0	3/5	0	4/5	2/5	0	52/5

Applying Dual Simplex.

		2	1	2	0	0	0	
	Basis	x_1	x_2	x_3	s_1	s_2	s_3	Sol
2	x_1	1	1/5	0	3/5	-1/5	0	19/5
2	x_3	0	3/5	1	-1/5	2/5	0	7/5
0	s_3	0	1/5	0	3/5	-1/5	1	-1/5
	Z	0	3/5	0	4/5	2/5	0	52/5
	Ratio	-	-	-	-	-1/2	-	

		2	1	2	0	0	0	
	Basis	x_1	x_2	x_3	s_1	s_2	s_3	Sol
2	x_1	1	0	0	0	0	-1	4
2	x_3	0	1	1	1	0	2	1
0	s_2	0	-1	0	-3	1	-5	1
	Z	0	1	0	2	0	2	10
	Ratio	-	-	-	-	-	-	

(2) Optimal and Feasible: $x_1 = 4$, $x_3 = 1$, $x_2 = 0$, $Z = 10$

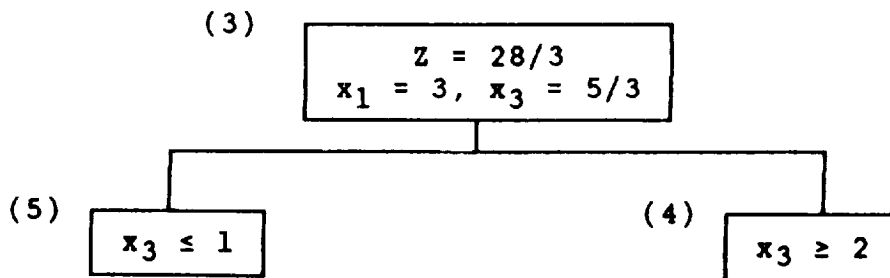
(3) Put $x_1 \leq 3$ into the tableau.

	2	1	2	0	0	0	
Basis	x_1	x_2	x_3	s_1	s_2	s_3	Sol
2	x_1	1	1/5	0	3/5	-1/5	19/5
2	x_3	0	3/5	1	-1/5	2/5	7/5
0	s_3	1	0	-1/5	0	0	3
	Z	0	3/5	0	4/5	2/5	52/5
	Ratio	-	-	-	-3/4	-	-

new s_3 row = $(1, 0, 0, 0, 0, 1, 3) - (1)(1, 1/5, 0, 3/5, -1/5, 0, 19/5)$

	2	1	2	0	0	0	
Basis	x_1	x_2	x_3	s_1	s_2	s_3	Sol
2	x_1	1	0	0	0	1	3
2	x_3	0	2/3	1	0	-1/3	5/3
0	s_1	0	1/3	0	1	-5/3	4/3
	Z	0	1/3	0	0	2/3	28/3

Optimal but noninteger -- Branch again!



(4) $x_3 \geq 2 \implies -x_3 \leq -2$

Basis	x_1	x_2	x_3	s_1	s_2	s_3	s_4	Sol
2 x_1	1	0	0	0	0	1	0	3
2 x_3	0	2/3	1	0	1/3	-1/3	0	5/3
0 s_1	0	1/3	0	1	-1/3	-5/3	0	4/3
0 s_4	0	0	-1	0	0	0	1	-2
Z	0	1/3	0	0	2/3	4/3	0	28/3
Ratio	-	-	-	-	-	-4	-	

New $s_4 \Rightarrow (0, 0, -1, 0, 0, 0, 1, -2) -$
 $(-1)(0, 2/3, 1, 0, 1/3, -1/3, 0, 5/3)$
 $= (0, 2/3, 0, 0, 1/3, -1/3, 1, -1/3)$

Basis	x_1	x_2	x_3	s_1	s_2	s_3	s_4	Sol
2 x_1	1	2	0	0	1	0	3	2
2 x_3	0	0	1	0	0	0	-1	2
0 s_1	0	-3	0	1	-2	0	-5	3
0 s_3	0	-2	0	0	-1	1	-3	1
Z	0	3	0	0	2	0	4	8

Feasible but $x_0 < x_0 @ (2)$.

Therefore, best is still $x_1 = 4, x_3 = 1, x_2 = 0, Z = 10$.

Using Branch (5) $x_3 \leq 1$ and using (3) as initial tableau we have

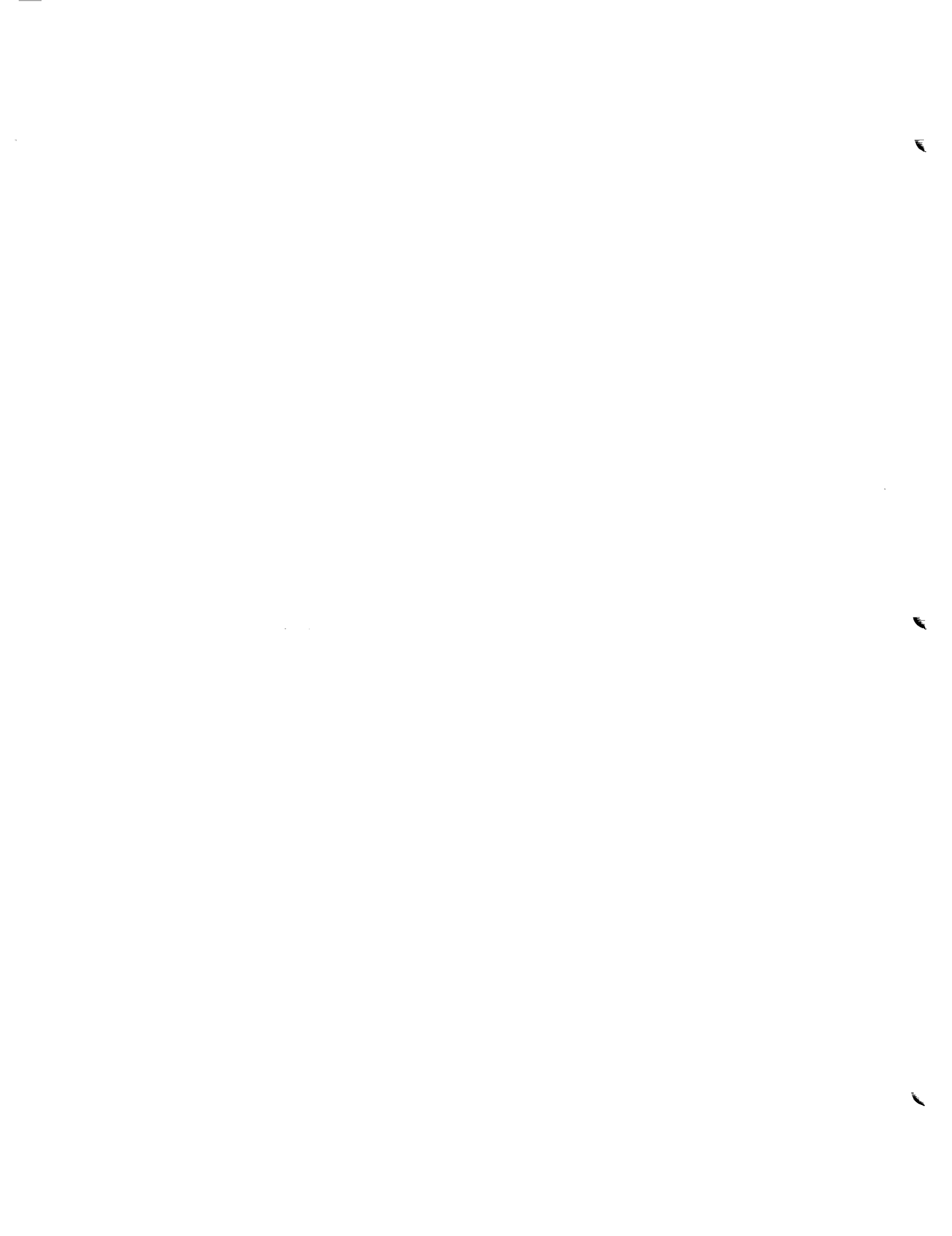
Basis	x_1	x_2	x_3	s_1	s_2	s_3	s_4	Sol
2 x_1	1	0	0	0	0	1	0	3
2 x_3	0	2/3	1	0	1/3	-1/3	0	5/3
0 s_1	0	1/3	0	1	-1/3	-5/3	0	4/3
0 s_4	0	0	1	0	0	0	1	1
Z	0	1/3	0	0	2/3	4/3	0	28/3
Ratio	-	$\frac{1/3}{-2/3}$	-	-	$\frac{2/3}{-1/3}$	-	-	

New $s_4 = (0, 0, 1, 0, 0, 0, 1, 1) - (1)(0, 2/3, 1, 0, 1/3, -1/3, 0, 5/3)$

Basis	x_1	x_2	x_3	s_1	s_2	s_3	s_4	Sol
2 x_1	1	0	0	0	0	1	0	3
2 x_3	0	0	1	0	0	0	1	1
0 s_1	0	0	0	1	-1/2	-3/2	-1/2	1
1 x_2	0	1	0	0	1/2	-1/2	-3/2	1
Z	0	0	0	0	1/2	3/2	1/2	9

Feasible solution $x_1 = 3, x_2 = 1, x_3 = 1, Z = 9$

But it still does not improve $Z = 10, x_1 = 4, x_2 = 0, x_3 = 1$. Therefore, optimal and Best Solution is $x_1 = 4, x_2 = 0, x_3 = 1, Z = 10$ (found with Branch (2)).



APPENDIX B
IESIP PASCAL COMPUTER CODE

```

PROGRAM IESIP;
CONST
{ The maximum number of variables plus 3. }
  MAXVAR = 33;
TYPE
{ Datarec will hold a node and pointers to the next
  and previous node on the linked list. }
  Recptr = ^Datarec;
  Datarec = Record
    Value : Array [1..MAXVAR] of Real;
    Previous : Recptr;
    Next : Recptr
  END;
  Inrec = Record
    Value : Array [1..80] of Char;
  END;
  Listptr = ^Listrec;
{ Listrec will hold pointers, one to a node entry, and
  one each to the next and previous elements on the
  linked list. }
  Listrec = Record
    Value : Recptr;
    Previous : Listptr;
    Next : Listptr;
  END;
VAR
{Boolean variables}
  Chkpar : Boolean;
  Cmderr : Boolean;
  Datafile : Boolean;
  Dispex : Boolean;
  Finish : Boolean;
  Maxmin : Boolean;
  Oddeven : Boolean;
  Opflag : Boolean;
  Parserr : Boolean;
  Parfound : Boolean;
  Shortcut : Boolean;
  Stype : Boolean;
{Integer variables}
  Calnum : Integer;
  Numex : Integer;
  Numline : Integer;
  Numofvar : Integer;
{Real variables}
  Numadd : Real;
  Numcomp : Real;
  Nummult : Real;
  Rdadd : Real;
  Rdcomp : Real;
  Rdmult : Real;
  Valex : Real;
{String and Char variables}
  Filename : String[32];

```

```

Filevar : Text;
Helpvar : Text;
Online  : String[80];
Yesno   : Char;
{Pointers variables}
Bestsol : Recptr;
Bestlist : Listptr;
Confun  : Recptr;
Consol  : Recptr;
Currcon : Recptr;
Datafun : Recptr;
Expfun  : Recptr;
Listelem : Listptr;
Listone  : Listptr;
Listtwo  : Listptr;
Lstelem  : Listptr;
Opfun   : Recptr;
Parlist  : Listptr;
Parseline: ^Inrec;
Startfun : Recptr;

```

```

PROCEDURE Writefun (Func : Recptr);
{ Write the node with the coefficients followed by the
variable x1 ... xN. This procedure is used only to
print the objective and constraint functions. If
printing a constraint function then print the operand
and the constraint value. }

```

```

VAR

```

```

    I : Integer;

```

```

BEGIN

```

```

    For I := 1 to Numofvar Do

```

```

        BEGIN

```

```

            If (Func^.Value[I] >= 0) then Write('+');

```

```

            Write(Func^.Value[I]:0:2);

```

```

            Write(' X');

```

```

            Write(I);

```

```

            Write(' ');

```

```

        END;

```

```

    If Not Opflag then

```

```

        BEGIN

```

```

            Case Trunc(Func^.Value[MAXVAR-1]) of

```

```

                1: Write('= ');

```

```

                2: Write('~= ');

```

```

                3: Write('< ');

```

```

                4: Write('<= ');

```

```

                5: Write('> ');

```

```

                6: Write('>= ');

```

```

            END;

```

```

            Write(Func^.Value[MAXVAR]:0:2)

```

```

        END;

```

```

        Writeln(' ')

```

```

    END;

```

```

PROCEDURE Writecon (Func : Recptr);
{ Write the rounded continuous solution in the form
(x1, x2, ..., xN). This procedure is used only to
print the rounded continuous solution. }
VAR
  I : Integer;
BEGIN
  Write('(');
  For I := 1 to Numofvar -1 Do
  BEGIN
    Write(Trunc(Func^.Value[I]));
    Write(', ');
  END;
  Write(Trunc(Func^.Value[Numofvar]));
  Writeln(')');
End;

PROCEDURE Writeeq (Func : Recptr);
{ Write the node in the form ( x1, x2, ..., xN) = value.
In the case where the node did not pass the
constraint's check, the value will be set to C.F. }
VAR
  I : Integer;
BEGIN
  Write('(');
  For I := 1 to Numofvar -1 Do
  BEGIN
    Write(Trunc(Func^.Value[I]));
    Write(', ');
  END;
  Write(Trunc(Func^.Value[Numofvar]));
  Write(') = ');
  If (Trunc(Func^.Value[MAXVAR-2]) = 1) then
    Writeln('C.F.')
  else
    Writeln(Func^.Value[MAXVAR]:0:3);
END;

PROCEDURE Printproblem;
{ Display the initial problem to the used in the form of
problem type, number of variables, objective function,
constraint function(s), and rounded continuous
solution. This display will occur if the initial
information comes from a data file, or at the users
request at the end of the program. }
BEGIN
  Writeln(' ');
  If Maxmin then
    Writeln('Maximization problem')
  else
    Writeln('Minimization problem');
  Writeln('Number of variables = ', Numofvar);
  Writeln(' ');
  Writeln('Objective function:');

```

```

Opflag := True;
Writefun(Opfun);
Writeln(' ');
Writeln('Constraint function(s):');
Opflag := False;
Datafun := Confun;
While Not (Datafun = Nil) Do
BEGIN
    Writefun(Datafun);
    Datafun := Datafun^.Next
END;
Writeln(' ');
Writeln('Continuous solution rounded value:');
Writecon(Consol);
Writeln(' ');
END;

PROCEDURE Openhelp;
BEGIN
{ Open the help file just in case the user needs it. }
    Assign(Helpvar, 'IESIP.HLP');
    Reset(Helpvar);
END;

PROCEDURE Readhelp(num : Integer);
{ Read the help file and look for the help section for a
given number. The help file is a simple ASCII text
file with a record #num that will denote the start of
the help section for that number. The help section
will then run until another #num record is
encountered. }
VAR
    Canprint : Boolean;
    Done      : Boolean;
    numon     : Integer;
BEGIN
    Reset(Helpvar);
    Done := False; Canprint := False;
    While Not Done Do
    BEGIN
        Read(Helpvar, Yesno);
        If (Yesno = '#') then
        BEGIN
            If Canprint then Done := True;
            Canprint := False;
            Readln(Helpvar, numon);
            If (numon = num) then Canprint := True;
        END
        Else
        BEGIN
            Readln(Helpvar, Online);
            If Canprint then
            BEGIN
                Write(Yesno);
            END
        END
    END

```

```

        Writeln(Online);
    END;
    END;
    Done := Done or Eof(Helpvar);
END;
END;

PROCEDURE Endprogram;
LABEL
    1;
VAR
    Compsc : Real;
BEGIN
    { If we have a data file open then close it. Ask the
      user if they wish to have the problem restated, if so
      then display the initial problem. Then list the best
      solution(s) found, and give computational results. }
    If Datafile then
        Close(Filevar);
    Writeln(' ');
1: Writeln('Do you wish to have the problem restated? ');
   Write('[y]es/[n]o/[h]elp default = no ');
   Readln(Yesno);
   If ((Yesno = 'y')or(Yesno = 'Y')) then
       Printproblem
   else If ((Yesno = 'h')or(Yesno = 'H')) then
       BEGIN
           Readhelp(20);
           Goto 1;
       END;
   Writeln(' ');
   Writeln('Best solution(s) found:');
   While Not (Bestlist = Nil) Do
       BEGIN
           Bestsol := Bestlist^.Value;
           Write (' ');
           Writeeq(Bestsol);
           Bestlist := Bestlist^.Next;
       END;
   Writeln(' ');
   Writeln('Computation of the rounded optimal
     solution required:');
   Write(' ');
   Write('Additions: ');
   Write(Trunc(Rdadd));
   Write(' Multiplications: ');
   Write(Trunc(Rdmult));
   Write(' Compares: ');
   Writeln(Trunc(Rdcomp));
   Writeln(' ');
   Write(Numex);
   Writeln(' Points were expanded requiring');
   Write(' ');
   Write('Additions: ');

```



```

Write(Trunc(Numadd));
Write(' Multiplications: ');
Write(Trunc(Nummult));
Write(' Compares: ');
Writeln(Trunc(Numcomp));
Writeln(' ');
Compsc := Rdcomp + Numcomp + Rdadd * 1.1666 +
  Numadd * 1.1666 + Rdmult * 4 + Nummult * 4;
Writeln('Computational complexity score = ', Compsc:0:3);
Writeln(' ');
END;

PROCEDURE Insertcon (Func : Recptr);
{ Insert the constraint function on to the
  constraint list. }
BEGIN
  If (Currcon = Nil) then
    BEGIN
      Confun := Func;
      Currcon := Func
    END
  else
    BEGIN
      Currcon^.Next := Func;
      Func^.Previous := Currcon;
      Func^.Next := Nil;
      Currcon := Func
    END;
END;

PROCEDURE Parse (Func : Recptr);
{ Parse a input string for the objective function or a
  constraint function. Start by setting all of the
  coefficients to zero then process the input string
  character by character in the following manner:
  (1) If the input character is a number, then multiply
  the number thusfar by ten and add the new number.
  (2) If the input character is a ; then get the next input
  string. This will allow function to span more than one
  line. (3) If the input character is an operand, then set
  the operand field of the node to that operand. If the
  input character is a X then set a flag to denote that we
  are now looking for the variable number. If the input
  character is a field separator space, +, or -, and we
  have the coefficient value and the variable number;
  then, store the coefficient in the variable. }
LABEL
  1;
VAR
  Num, Value, Exp, Ten, Unt : Real;
  I, J : Integer;
  C : Char;
  Xfound : Boolean;
BEGIN

```

```

For I := 1 to MAXVAR do
  Func^.Value[I] := 0;
  Num := 0;
  Xfound := false;
  Exp := 1;
  Ten := 10.0;
  Unt := 1.0;
1: For I := 2 to Length(Online)+1 Do
  BEGIN
    C := Parseline^.Value[I];
    Case C of
      '0' .. '9': BEGIN
        If (Num = 0.0) then
          Num := Exp * (Ord(c) - Ord('0')) * Unt
        else
          Num := Exp * Num * Ten + (Ord(C) -
            Ord('0')) * Unt;
        If Not (Num = 0) then Exp := 1;
        If (Unt < 1) then Unt := Unt / 10.0;
      END;
      ';': BEGIN
        If Datafile then
          Readln(Filevar, Online)
        else
          Readln(Online);
        Goto 1
      END;
      'x', 'X': If Not Xfound then
        BEGIN
          Xfound := True;
          Value := Num;
          Num := 0;
          Ten := 10.0;
          Unt := 1.0;
        END
      else
        BEGIN
          Parserr := True;
          I := 80
        END;
      '~': Func^.Value[MAXVAR-1] := 1;
      '=': Func^.Value[MAXVAR-1] :=
        Func^.Value[MAXVAR-1] + 1;
      '<': Func^.Value[MAXVAR-1] := 3;
      '>': Func^.Value[MAXVAR-1] := 5;
      ' ', '+': BEGIN
        Exp := 1;
        If Xfound then
          If (Num > 0) then
            BEGIN
              J := Trunc(Num);
              Func^.Value[J] := Value;
              Num := 0;
              Xfound := False
            END
          END
        END
      END;
    END;
  END;

```

```

                                END
                                else
                                BEGIN
                                    Parserr := True;
                                    I := 80
                                END;
                                END;
                                '-':
                                BEGIN
                                    Exp := -1;
                                    If Xfound then
                                        If (Num > 0) then
                                            BEGIN
                                                J := Trunc(Num);
                                                Func^.Value[J] := Value;
                                                Num := 0;
                                                Xfound := False
                                            END
                                        else
                                            BEGIN
                                                Parserr := True;
                                                I := 80
                                            END;
                                        END;
                                END;
                                '.':
                                BEGIN
                                    Ten := 1;
                                    Unt := 0.1;
                                END;
                                else
                                BEGIN
                                    I := 80;
                                    If Xfound then
                                        If (Num > 0) then
                                            BEGIN
                                                J := Trunc(num);
                                                Func^.Value[J] := Value
                                            END
                                        else
                                            Parserr := True;
                                        END;
                                END;
                                END; { Case }
                                END; { For loop }
                                If Xfound then
                                    If (Num > 0) then
                                        BEGIN
                                            J := Trunc(Num);
                                            Func^.Value[J] := Value;
                                            Num := 0;
                                            Xfound := False;
                                        END
                                    else
                                        Parserr := True;
                                    If (Func^.Value[MAXVAR-1] > 0) then
                                        Func^.Value[MAXVAR] := Num
                                    END;
                                END;

```

```

FUNCTION Evalfun (Fun1, Fun2 : Recptr) : Real;
{ Evaluate a function; we will receive two nodes, one will
  contain the coefficients for the variables and the other
  will contain the value of the variables. Multiply the
  coefficient by the value for each variable and sum the
  result. }
VAR
  I : Integer;
  Num : Real;
BEGIN
  Num := 0;
  For I := 1 to Numofvar Do
    Num := Num + Fun1^.value[I] * Fun2^.Value[I];
  Numadd := Numadd + Numofvar - 1;
  Nummult := Nummult + Numofvar;
  Evalfun := Num;
END;

PROCEDURE Roundcon;
{ Round the continuous solution. The first part of this
  procedure will parse the continuous solution using the
  same rules for numbers and ; as the previous parse. The
  field separator will be a , and cause the value of the
  variable to be stored. Once the continuous solution has
  been parsed, the solution will be rounded using the
  method stated in chapter III of the dissertation. }
LABEL
  1,2;
VAR
  Num, Ten, Unt, Exp : Real;
  Rvalue : Array [1..MAXVAR] of Real;
  I,J,K : Integer;
  C : Char;
  Pass : Boolean;
BEGIN
  J := 0;
  Num := 0.0;
  Ten := 10.0;
  Unt := 1.0;
  Exp := 1.0;
1: For I:= 3 to 80 Do
  BEGIN
    C := Parseline^.Value[I];
    Case C of
      '0' .. '9': BEGIN
          If (Num = 0.0) then
            Num := Exp * (Ord(c)-Ord('0'))*Unt
          else
            Num := Exp * Num * Ten +
              (Ord(C)-Ord('0'))*Unt;
          If Not (Num = 0.0) then Exp := 1.0;
          If (Unt < 1) then Unt := Unt / 10.0
        END;
      ',': BEGIN

```

```

        Ten := 1.0;
        Unt := 0.1
    END;
    ')',',': BEGIN
        J := J + 1;
        Rvalue[J] := Num;
        Num := 0.0;
        Ten := 10.0;
        Unt := 1.0;
        If (C = ')') then I := 80;
    END;
    ';': BEGIN
        If Datafile then
            Readln(Filevar, Online)
        else
            Readln(Online);
        Goto 1
    END;
    ' ': I := i;
else BEGIN
    Parserr := True;
    I := 80
END;
END; { Case}
END; {For loop}
If (J < Numofvar) then Parserr := True;
If Not Parserr then
BEGIN
    New(consol);
    For I := 1 to Numofvar Do
        If Not Maxmin then
            Consol^.Value[I] := Trunc(Rvalue[I]) + 1
        else if (Rvalue[I] = Int(Rvalue[I])) then
            Consol^.Value[I] := Trunc(Rvalue[I]) - 1
        else
            Consol^.Value[I] := Trunc(Rvalue[I]);
    For I := 1 to Numofvar Do
        If Consol^.Value[I] < 0 then
            Consol^.Value[I] := 0;
2: Datafun := Confun;
Repeat
    Num := Evalfun (consol, Datafun);
    Numcomp := Numcomp + 1;
    Pass := True;
    Case Trunc(Datafun^.Value[MAXVAR-1]) of
    1: If Not(Num=Datafun^.Value[MAXVAR]) then
        Pass := False;
    2: If (Num=Datafun^.Value[MAXVAR]) then
        Pass := False;
    3: If (Num>=Datafun^.Value[MAXVAR]) then
        Pass := False;
    4: If (Num>Datafun^.Value[MAXVAR]) then
        Pass := False;
    5: If (Num<=Datafun^.Value[MAXVAR]) then

```

C-2

```

        Pass := False;
    6: If (Num<Datafun^.Value[MAXVAR]) then
        Pass := False;
END; { Case }
If Not Pass then
BEGIN
    J := 1;
    Num := Datafun^.Value[1];
    For I := 2 to Numofvar Do
    BEGIN
        If Maxmin then
        BEGIN
            If (Datafun^.Value[I] > Num) then
            BEGIN
                J := I;
                Num := Datafun^.Value[I]
            END;
        END
        else
        BEGIN
            If (Datafun^.Value[I] < Num) then
            BEGIN
                J := I;
                Num := Datafun^.Value[I]
            END;
        END
    END;
    If Maxmin then
        Consol^.Value[J] := Consol^.Value[J] -1
    else
        Consol^.Value[J] := Consol^.Value[J] + 1;
    Num := 0.0;
    For I := 1 to Numofvar do
        Num := Num + Consol^.Value[I];
    If Num > 0 then
        Goto 2
    else
        Parserr := True;
    END;
    Datafun := Datafun^.Next;
    If Parserr then
        Datafun := Nil;
    Until (Datafun = Nil)
    END { No Parse error}
END;

PROCEDURE Initialize;
LABEL
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12;
VAR
    Done : Boolean;
BEGIN
    Openhelp;
    { Find the type of problem that the user wishes to solve. }

```

```

1: Writeln('Is the objective function to be maximized? ');
   Write('[y]es/[n]o/[h]elp default = no ');
   Readln(Yesno);
   If ((Yesno = 'y')or(Yesno = 'Y')) then
       Maxmin := True
   else If ((Yesno = 'h')or(Yesno = 'H')) then
       BEGIN
           Readhelp (1);
           Goto 1
       END
   else
       Maxmin := False;
{ Find out if the data input is coming from a data file
  or the screen. }
2: Writeln('Do you have the functions in a data file? ');
   Write('[y]es/[n]o/[h]elp default = no ');
   Readln(Yesno);
   If ((Yesno = 'y')or(Yesno = 'Y')) then
       Datafile := True
   else If ((Yesno = 'h')or(Yesno = 'H')) then
       BEGIN
           Readhelp (2);
           Goto 2
       END
   else
       Datafile := False;
{ If we have a data file, then open the file for input. }
3: If Datafile then
   BEGIN
{NOTE: When using IBM DOS, the assign and reset commands
  can be used to open any given file. This will be
  machine dependent. }
       Write('Enter the file? ');
       Readln(Filename);
       Assign(Filevar, Filename);
       Reset(Filevar)
   END;
{ Get the number of variables. }
   If Datafile then
       Readln(Filevar, Numofvar)
   else
       BEGIN
           Write('How many variables are in this problem? ');
           Readln(Numofvar)
       END;
       Parseline := Addr(Online);
       Parserr := False;
{ Get the objective function. }
5: If Datafile then
       Readln(Filevar, Online)
   else
       BEGIN
           Writeln('Enter the Objective Function/[h]elp ');
           Readln(Online)
       END

```

```

END;
If ((Online = 'h')or(Online ='H')) then
BEGIN
  Readhelp(5);
  Goto 5;
END;
New(Opfun);
Opflag := True;
Parserr := False;
Parse(Opfun);
Done := Parserr;
{ Get the Constraint function(s). }
Opflag := False;
Confun := Nil;
Currcon := Nil;
6: Repeat
  If Datafile then
    Readln(Filevar, Online)
  else
    BEGIN
      Writeln('Enter the constraint
              function/[h]elp/[q]uit ');
      Readln(Online)
    END;
    If ((Online='h')or(Online='H')) then
    BEGIN
      Readhelp(6);
      Goto 6;
    END;
    If (Parseline^.Value[2] = '(') then Done := True;
    If ((Parseline^.Value[2]='q')or
        (Parseline^.Value[2]='Q')) then Done := True;
    If Not Done then
    BEGIN
      New(Datafun);
      Parse(Datafun);
      Insertcon(Datafun)
    END;
  Until Done or Parserr;
  If Parserr then goto 8;
  If (Parseline^.Value[2]='(') then goto 8;
7: Writeln('Enter the optimal continuous
          solution/[h]elp ');
  Readln(Online);
  If ((Online='h')or(Online='H')) then
  BEGIN
    Readhelp(7);
    Goto 7
  END;
8: Numadd := 0;
   Nummult := 0;
   Numcomp := 0;
   If Not Parserr then Roundcon;
   If Parserr then

```



```

BEGIN
  Writeln(' Could not find a rounded solution,
          infeasibility likely. ');
  Goto 12;
END;
If Datafile then
  Printproblem
else
  BEGIN
    Writeln('Rounded optimal continuous solution');
    Writecon(Consol)
  END;
9: Writeln('Do you wish to accept this rounded solution?');
   Write('[y]es/[n]o/[h]elp default = yes ');
   Readln(Yesno);
   If ((Yesno = 'h')or(Yesno='H')) then
     BEGIN
       Readhelp(9);
       Goto 9
     END
   else if ((Yesno = 'n')or(Yesno = 'N')) then
     BEGIN
       Dispose(Consol);
       Writeln('Enter your rounded solution');
       Readln(Online);
       Roundcon
     END;
10: Writeln('Enter the number of initial expansions');
     Writeln('Integer or 0 for help');
     Readln(Calnum);
     If (Calnum < 1) then
       BEGIN
         Readhelp(10);
         Goto 10
       END;
11: Writeln('Do you wish to see the nodes as they
          are expanded? ');
     Write('[y]es/[n]o/[h]elp default = .no ');
     Readln(Yesno);
     If ((Yesno = 'h')or(Yesno='H')) then
       BEGIN
         Readhelp(11);
         Goto 11
       END
     else if ((Yesno = 'y')or(Yesno = 'Y')) then
       Dispex := True
     else
       Dispex := False;
{ Set all of the initial values. }
  Parfound := False;
  Shortcut := False;
  Stype := False;
  New(Listone);
  Listone^.Value := Consol;

```

```

Listone^.Next := Nil;
Listtwo^.Previous := Nil;
Listtwo := Nil;
Parlist := Nil;
Oddeven := False;
Finish := Parserr;
Rdadd := Numadd;
Rdmult := Nummult;
Rdcomp := Numcomp;
Numadd := 0;
Nummult := 0;
Numcomp := 0;
Numex := 0;
Numline := 0;
Startfun := Consol;
Startfun^.Value[MAXVAR] := Evalfun(Startfun, Opfun);
Startfun^.Value[MAXVAR-2] := 0.0;
Startfun^.Next := Nil;
Startfun^.Previous := Nil;
12: Bestsol := Startfun;
   New(Bestlist);
   Bestlist^.Value := Bestsol;
   Bestlist^.Next := Nil;
   Bestlist^.Previous := Nil;
END;

PROCEDURE Copyvar;
{ Copy the value in the node Datafun to the node Expfun. }
VAR
  I : Integer;
BEGIN
  For I := 1 to Numofvar Do
    Expfun^.Value[I] := Datafun^.Value[I];
  Expfun^.Value[MAXVAR-2] := 0;
  Expfun^.Value[MAXVAR-1] := 0;
  Expfun^.Value[MAXVAR] := 0;
END;

FUNCTION Eqfun (Fun1, Fun2 : Recptr) : Boolean;
{ Compare two nodes if they have equal values for all
  variables then return true else false. }
VAR
  I : Integer;
BEGIN
  Eqfun := True;
  For I := 1 to Numofvar do
    If Not (Fun1^.Value[I] = Fun2^.Value[I]) then
      Eqfun := False;
END;

FUNCTION Conchk (Fun1 : Recptr) : Real;
{ Compare a node with the constraints. If the node
  passes all of the constraints then return a zero;
  if not, then return a one. }

```

```

VAR
  Num : Real;
  Fun : Recptr;
BEGIN
  Conchk := 0;
  Fun := Confun;
  While Not (Fun = Nil) DO
    BEGIN
      Numcomp := Numcomp + 1;
      Num := Evalfun (Fun1, Fun);
      Case Trunc(Fun^.Value[MAXVAR-1]) of
        1: If Not (Num=Fun^.Value[MAXVAR]) then
            Conchk := 1.0;
        2: If (Num=Fun^.Value[MAXVAR]) then
            Conchk := 1.0;
        3: If (Num>=Fun^.Value[MAXVAR]) then
            Conchk := 1.0;
        4: If (Num>Fun^.Value[MAXVAR]) then
            Conchk := 1.0;
        5: If (Num<=Fun^.Value[MAXVAR]) then
            Conchk := 1.0;
        6: If (Num<Fun^.Value[MAXVAR]) then
            Conchk := 1.0;
      END;
      Fun := Fun^.Next;
    END;
  END;

FUNCTION Searchpar(Fun1 : Recptr) : Boolean;
{ Search the parent list to see if a node is present.
  All nodes are unique, that is if we generate a node
  and then later generate the same node then we will
  detect this fact and reset to pointer to the first
  node that was generated. The parent list is a list
  of pointers to nodes that have been expanded, therefore
  if we have the same pointer as a member of the list
  then we have a parent point. }
VAR
  Lstelem : Listptr;
BEGIN
  Searchpar := False;
  Lstelem := Parlist;
  While Not (Lstelem = Nil) do
    BEGIN
      If (Lstelem^.Value = Fun1) then Searchpar := True;
      Lstelem := Lstelem^.Next;
    END;
  END;

PROCEDURE Insertlist( Fun4 : Recptr);
{ Insert the node on the list of nodes to be expanded.
  That is, we use two lists, one is the nodes currently
  being expanded and the other is the nodes that will
  be expanded next time. Also, when inserting the node,

```

maintain the order of the node to be expanded. For a maximization problem the order is decreasing and for minimization the order is increasing. }

```

VAR
  Fun1 : Listptr;
  Fun2 : Recptr;
  Fun3 : Listptr;
  Fun5 : Listptr;
  Done : Boolean;
BEGIN
  If Oddeven then
    Fun1 := Listone
  else
    Fun1 := Listtwo;
  Done := False;
  Fun3 := Nil;
  While Not (Fun1 = Nil) and Not Done do
    BEGIN
      Fun2 := Fun1^.Value;
      If ((Fun4^.Value[MAXVAR] > Fun2^.Value[MAXVAR])
        and Maxmin) or ((Fun4^.Value[MAXVAR] <
        Fun2^.Value[MAXVAR]) and Not Maxmin) then
        BEGIN
          New(Fun5);
          Fun5^.Value := Fun4;
          Fun5^.Next := Fun1;
          Fun5^.Previous := Fun3;
          If Not (Fun3 = Nil) then
            Fun3^.Next := Fun5
          else
            BEGIN
              If Oddeven then
                Listone := Fun5
              else
                Listtwo := Fun5;
            END;
          If Not (Fun1 = Nil) then
            Fun1^.Previous := Fun5;
          Done := True;
        END
      else if (Fun4 = Fun2) then Done := True
      else
        BEGIN
          Fun3 := Fun1;
          Fun1 := Fun1^.Next;
        END;
    END;
  If Not Done then
    BEGIN
      New(Fun5);
      Fun5^.Value := Fun4;
      Fun5^.Next := Nil;
      Fun5^.Previous := Fun3;
      If Not (Fun3 = Nil) then

```

```

        Fun3^.Next := Fun5
    else
    BEGIN
        If Oddeven then
            Listone := Fun5
        else
            Listtwo := Fun5;
        END;
    END;
END;

PROCEDURE Insertfun;
{ Insert the node on to a master list of all nodes
generated in decreasing order. This procedure will
also determine if the node has been generated before
if so, it will reset the pointer to that first
occurrence. If the node has not been generated before,
then it will be evaluated with the constraint function(s)
to insure that it passes all constraints. Also,
it will determine if the node should be placed on the
list to be expanded during the next pass. }
LABEL
1;
VAR
    Onelst : Recptr;
    Twolst : Recptr;
BEGIN
    Onelst := Startfun;
    Twolst := Nil;
    While not (Onelst = Nil) and
        (Onelst^.Value[MAXVAR] > Expfun^.Value[MAXVAR]) Do
    BEGIN
        Twolst := Onelst;
        Onelst := Onelst^.Next;
    END;
1: If Onelst = Nil then
    BEGIN
        Expfun^.Value[MAXVAR-2] := Conchk (Expfun);
        Expfun^.Previous := Twolst;
        Expfun^.Next := Nil;
        Twolst^.Next := Expfun;
    END
    else if (Onelst^.Value[MAXVAR] <
        Expfun^.Value[MAXVAR]) then
    BEGIN
        Expfun^.Value[MAXVAR-2] := Conchk (Expfun);
        Expfun^.Next := Onelst;
        Expfun^.Previous := Twolst;
        Onelst^.Previous := Expfun;
        If Not (Twolst = Nil) then
            Twolst^.Next := Expfun
        else
            Startfun := Expfun;
        END
    END
END

```

```

else
BEGIN
  If Not Eqfun(Onelst, Expfun) then
  BEGIN
    Twolst := Onelst;
    Onelst := Onelst^.Next;
    Goto 1;
  END;
  Dispose(Expfun);
  Expfun := Onelst;
END;
If Trunc(Expfun^.Value[MAXVAR-2]) = 0 then
BEGIN
  If Not Parfound then
    Insertlist(Expfun)
  else if Not Searchpar(Expfun) then
    if (Expfun^.Value[MAXVAR] = Valex) then
      Insertlist(Expfun);
END;
END;

PROCEDURE Stopline;
{ Stop and hold the display until the user enters
  a return to continue. }
BEGIN
  Write('Press RETURN to continue...');
  Readln;
  Writeln(' ');
END;

FUNCTION Searchlist (Funl : Recptr) : Boolean;
{ Once we have found a parent we will have to evaluate
  each of the nodes we are about to expand to determine the
  value that will cause the next node to be expanded. This
  procedure will take the nodes that will be generated
  and check to see if they are on the master list or not.
  Next the procedure will determine if the nodes pass
  the constraints, thereby allowing us to determine
  the cut off value of the nodes that will be
  expanded next time. }
LABEL
  1, 2;
VAR
  Onelst, Twolst : Recptr;
  Pass : Boolean;
BEGIN
  Searchlist := True;
  Onelst := Startfun;
  Twolst := Funl;
  While Not (Onelst = Nil) and
    (Onelst^.Value[MAXVAR] > Twolst^.Value[MAXVAR]) Do
    Onelst := Onelst^.Next;
1: If Onelst = Nil then Goto 2
   else If (Onelst^.Value[MAXVAR] <

```

```

                Twolst^.Value[MAXVAR]) then Goto 2;
If Not Eqfun(Onelst, Twolst) then
BEGIN
    Onelst := Onelst^.Next;
    Goto 1;
END
else
    Twolst := Onelst;
2: Pass := Not Searchpar(Twolst);
If Pass then
    Twolst^.Value[MAXVAR-2] := Conchk(Twolst);
If (Twolst^.Value[MAXVAR-2] = 1.0) then
    Pass := False;
Searchlist := Pass;
END;

FUNCTION Parval (Datafun : Recptr) : Real;
{ Determine what is the value that will cause the maximum
  gain for the next expansion. That is the node generated
  must not be a parent and must pass all constraints. This
  will be determined by searchlist. We then will determine
  the value that will cause the maximum gain. A maximum
  gain must be an improvement over the root node. In
  the case where there is more than one node that generates
  an improvement, we will take the best improvement. }
VAR
    Num, Val : Real;
    Fir, Pass : Boolean;
    I : Integer;
BEGIN
    New(Expfun);
    Fir := True;
    Num := 0;
    For I := 1 to Numofvar Do
        BEGIN
            Copyvar;
            If Maxmin then
                Expfun^.Value[I] := Datafun^.Value[I] + 1
            else
                Expfun^.Value[I] := Datafun^.Value[I] - 1;
            If (Expfun^.Value[I] >= 0) then
                BEGIN
                    Val := Evalfun(Expfun, Opfun);
                    Expfun^.Value[MAXVAR] := Val;
                    Pass := Searchlist(Expfun);
                    If Pass then
                        If Fir then
                            BEGIN
                                Num := Val;
                                Fir := False;
                            END
                        else
                            BEGIN
                                If ((Val > Num) and Maxmin) or

```

```

                ((Val < Num) and Not Maxmin) then
                    Num := Val;
            END;
        END;
    Copyvar;
    If Maxmin then
        Expfun^.Value[I] := Datafun^.Value[I] - 1
    else
        Expfun^.Value[I] := Datafun^.Value[I] + 1;
    If (Expfun^.Value[I] >= 0) then
        BEGIN
            Val := Evalfun(Expfun, Opfun);
            Expfun^.Value[MAXVAR] := Val;
            Pass := Searchlist(Expfun);
            If Pass then
                If Fir then
                    BEGIN
                        Num := Val;
                        Fir := False;
                    END
                else
                    BEGIN
                        If ((Val > Num) and Maxmin) or
                            ((Val < Num) and Not Maxmin) then
                            Num := Val;
                        END;
                    END;
                END;
            END;
        Dispose(Expfun);
        Parval := Num;
    END;

PROCEDURE Newbest;
{ If the current node being expanded is a better solution
  to the problem, then reset the best solution list to
  just that node. }
VAR
    Onelist : Listptr;
BEGIN
    While Not (Bestlist = Nil) Do
        BEGIN
            Onelist := Bestlist^.Next;
            Dispose(Bestlist);
            Bestlist := Onelist;
        END;
    New(Bestlist);
    Bestlist^.Value := Datafun;
    Bestlist^.Next := Nil;
    Bestlist^.Previous := Nil;
    Bestsol := Datafun;
END;

PROCEDURE Addbest;
{ If the current node being expanded has the same

```



```

    value as the best solution, then add it to the best
    solution list. }
LABEL
    l;
VAR
    Onelist : Listptr;
BEGIN
    Onelist := Bestlist;
    While not (Onelist = Nil) do
        BEGIN
            If (Onelist^.Value = Datafun) then Goto l;
            Onelist := Onelist^.Next;
        END;
        New(Onelist);
        Onelist^.Value := Datafun;
        Onelist^.Next := Bestlist;
        Onelist^.Previous := Nil;
        Bestlist^.Previous := Onelist;
        Bestlist := Onelist;
    l: Bestlist := Bestlist;
END;

PROCEDURE Expandlist;
{ This is the main work procedure, where the node is
  expanded and placed on all the appropriate lists.
  First, the node to be expanded is compared with the
  best solution, if it is an improvement, then the best
  solution list is set to that node. If it is the same
  as the best solution, then the node is added to the
  best solution list. Next, if we have already found our
  first parent then determine the value that will cause an
  improvement. The node is univariately expanded to
  identify all possible neighboring solutions. These
  node values are copied into a generated node and
  checked against the constraints. That is, evaluate the
  node, insure that it passes the constraints, insert it
  onto the master list of nodes and if it passes the test,
  place it on the list of nodes to be expanded next time.
  Once we have a node expanded place it on
  the parent list so it will not be expanded again. }
VAR
    Done : Boolean;
    I : Integer;
BEGIN
    If Oddeven then
        Listelem := Listtwo
    else
        Listelem := Listone;
    Done := (Listelem = Nil);
    While Not Done Do
        BEGIN
            Datafun := Listelem^.Value;
            Numex := Numex + 1;
            If (Bestsol^.Value[MAXVAR] =

```

```

        Datafun^.Value[MAXVAR]) then Addbest;
If Maxmin then
    If (Bestsol^.Value[MAXVAR] <
        Datafun^.Value[MAXVAR]) then Newbest;
If Not Maxmin then
    If (Bestsol^.Value[MAXVAR] >
        Datafun^.Value[MAXVAR]) then Newbest;
If Dispex then
BEGIN
    Writeln(' ');
    Writeln(' ');
    Write('Node Expansion (');
    Write(Numex);
    Writeln(')');
    Write(' ');
    Numline := Numline + 3;
    Writeeq(Datafun)
END;
If Parfound then
BEGIN
    Valex := Parval(Datafun);
    If ((Datafun^.Value[MAXVAR] > Valex) and Maxmin) or
        ((Datafun^.Value[MAXVAR] < Valex) and
        Not Maxmin) then
        Valex := Datafun^.Value[MAXVAR];
END;
For I := 1 to Numofvar Do
BEGIN
    If Dispex and (Numline > 20) then
    BEGIN
        Stopline;
        Numline := 0;
    END;
    New(Expfun);
    Copyvar;
    If Maxmin then
        Expfun^.Value[I] := Datafun^.Value[I] + 1
    else
        Expfun^.Value[I] := Datafun^.Value[I] - 1;
    If Not(Expfun^.Value[I] < 0) then
    BEGIN
        Expfun^.Value[MAXVAR] := Evalfun(Expfun, Opfun);
        Insertfun;
        If Dispex then
        BEGIN
            Write(' ');
            Writeeq(Expfun);
            Numline := Numline + 1;
        END
    END;
    If Not Shortcut or
        (Trunc(Expfun^.Value[MAXVAR-2]) = 1) then
    BEGIN
        New(Expfun);

```

```

Copyvar;
If Maxmin then
  Expfun^.Value[I] := Datafun^.Value[I] - 1
else
  Expfun^.Value[I] := Datafun^.Value[I] + 1;
If Not (Expfun^.Value[I] < 0) then
BEGIN
  Expfun^.Value[MAXVAR] :=
    Evalfun(Expfun, Opfun);
  Insertfun;
  If Dispex then
  BEGIN
    Write(' ');
    Writeeq(Expfun);
    Numline := Numline + 1;
  END
END
END;
If Dispex and (Numline > 20) then
BEGIN
  Stopleveline;
  Numline := 0;
END;
END;
Lstelem := Lstelem^.Next;
If Not (Parlist = Nil) then
  Parlist^.Previous := Lstelem;
Lstelem^.Next := Parlist;
Lstelem^.Previous := Nil;
Parlist := Lstelem;
Lstelem := Lstelem;
Done := (Lstelem = Nil) or Stype;
If Stype then
  If Not (Lstelem = Nil) then
  BEGIN
    Expfun := Lstelem^.Value;
    If (Expfun^.Value[MAXVAR] =
      Datafun^.Value[MAXVAR]) then
      Done := False;
    END;
  If ((3+Numline+2*Numofvar) > 20) and Dispex then
  BEGIN
    Numline := 0;
    Stopleveline;
  END;
END;
While Not (Lstelem = Nil) Do
BEGIN
  Lstelem := Lstelem^.Next;
  Dispose(Lstelem);
  Lstelem := Lstelem;
END;
If Oddeven then
  Listtwo := Nil

```

```

    else
        Listone := Nil;
END;

FUNCTION Allpos(Func : Recptr) : Boolean;
{ Determine if all the coefficients of a function are
  positive, if so then return true. }
VAR
    I : Integer;
BEGIN
    Allpos := True;
    For I := 1 to Numofvar Do
        If (Func^.Value[I] < 0) then Allpos := False;
    END;

PROCEDURE Switchlist;
{ A flag oddeven is used to determine which list is to
  be expanded and which list is to be generated. This
  procedure will flip the flag, and thereby the list. }
BEGIN
    If Oddeven then
        Oddeven := False
    else
        Oddeven := True;
    END;

PROCEDURE Movelist;
{ Once we have determined that the top node to be expanded
  is the first parent we need to remove all other parent
  nodes from the list. }
VAR
    Lstelem : Listptr;
    Fun1 : Listptr;
    Fun2 : Recptr;
BEGIN
    If Oddeven then
        Lstelem := Listtwo
    else
        Lstelem := Listone;
    Lstelem := Nil;
    While Not (Lstelem = Nil) do
        BEGIN
            Fun2 := Lstelem^.Value;
            If Not Searchpar(Fun2) then
                BEGIN
                    If Lstelem = Nil then
                        BEGIN
                            New(Lstelem);
                            Lstelem^.Value := Fun2;
                            Lstelem^.Next := Nil;
                            Lstelem^.Previous := Nil;
                            If Oddeven then
                                Listone := Lstelem
                            else

```

```

                Listtwo := Lstelem;
            END
            else
            BEGIN
                New(Fun1);
                Fun1^.Value := Fun2;
                Fun1^.Previous := Lstelem;
                Fun1^.Next := Nil;
                Lstelem^.Next := Fun1;
                Lstelem := Fun1;
            END;
            END;
            Funl := Lstelem^.Next;
            Dispose(Lstelem);
            Lstelem := Funl;
        END;
        If Oddeven then
            Listtwo := Nil
        else
            Listone := Nil;
        Switchlist;
    END;

```

PROCEDURE Solveproblem;

{ The main control procedure. This procedure can be divided into three main sections. The first section will expand the list using a type I search for the initial number of times the user requested. The next section will set the flags so only the first node is expanded and continue that expansion until the first node on the list is a parent. The parents on the remainder of the list will be removed. The last section will expand the list until there is not a list to be expanded. }

LABEL 1;

VAR

I : Integer;

BEGIN

Chkpar := False;

For I := 0 to Calnum DO

BEGIN

Expandlist;

Switchlist;

If Oddeven then

Listelem := Listtwo

else

Listelem := Listone;

If (Listelem = Nil) then Goto 1;

END;

Chkpar := True;

Stype := True;

Shortcut := Allpos(Opfun);

If Oddeven then

Datafun := Listtwo^.Value

```

else
  Datafun := Listone^.Value;
While Not Parfound Do
BEGIN
  Expandlist;
  Switchlist;
  If Oddeven then
    Listelem := Listtwo
  else
    Listelem := Listone;
  If (Listelem = Nil) then Goto 1;
  Datafun := Listelem^.Value;
  Parfound := Searchpar(Datafun);
END;
Stype := False;
Movelist;
If Oddeven then
  Listelem := Listtwo
else
  Listelem := Listone;
Shortcut := False;
1: While not (Listelem = Nil) do
BEGIN
  Expandlist;
  Switchlist;
  If Oddeven then
    Listelem := Listtwo
  else
    Listelem := Listone;
END;
END;

BEGIN
{ The main procedure first initializes the problem.
  If we did not have an error in the
  initialization phase then it solves the problem
  and prints the results. }
  Initialize;
  If Not Parserr then
  BEGIN
    Solveproblem;
    Endprogram;
  END
  else
    Writeln('ERROR in input string');
END.

```

APPENDIX C
TEST PROBLEMS

Problem # 1 [Claycombe and Sullivan, p. 194]

$$\begin{aligned} \max & \quad 4,000x_1 + 7,000x_2 \\ \text{subject to} & \quad 1,200x_1 + 2,000x_2 \leq 6,000 \\ & \quad 25,000x_1 + 8,000x_2 \leq 20,000 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (1.739, 1.956)

Integer Solution: (5, 0)

Problem # 2 [Taha, 1975, p. 175]

$$\begin{aligned} \max & \quad x_1 + 2x_2 \\ \text{subject to} & \quad x_1 + x_2 \leq .9 \\ & \quad -2x_1 - x_2 \leq .2 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (0, 0.9)

Integer Solution: (0, 0)

Problem # 3 [Taha, 1987, p. 338]

$$\begin{aligned} \max & \quad 2x_1 + x_2 \\ \text{subject to} & \quad 10x_1 + 10x_2 \leq 9 \\ & \quad 10x_1 + 5x_2 \geq 1 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (0.9, 0)

Integer Solution: (infeasible)

Problem # 4 [Taha, 1987, p. 86]

$$\begin{aligned} \max & \quad 2x_1 + 4x_2 \\ \text{subject to} & \quad x_1 + 2x_2 \leq 5 \\ & \quad x_1 + x_2 \leq 4 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (0, 2.5)

Integer Solution: (1, 2)

Problem # 5 [Zionts, p. 485]

$$\begin{aligned} \max & \quad 5x_1 + 2x_2 \\ \text{subject to} & \quad 2x_1 + 2x_2 \leq 9 \\ & \quad 3x_1 + x_2 \leq 11 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (3.25, 1.25)

Integer Solution: (3, 1)

Problem # 6 [Hillier and Lieberman, p. 742]

$$\begin{aligned} \min & \quad 15x_1 + 10x_2 \\ \text{subject to} & \quad 3x_1 + x_2 \geq 6 \\ & \quad x_1 + x_2 \geq 3 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (1.5, 1.5)

Integer Solution: (2, 1)

Problem # 7 [Salkin, p. 120]

$$\begin{aligned} \min & \quad x_1 + x_2 \\ \text{subject to} & \quad x_1 + 2.5x_2 \geq 3 \\ & \quad x_1 + .4x_2 \geq 1.2 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (0.8571, 0.8571)

Integer Solution: (1, 1)

Problem # 8 [Llewellyn, p. 271]

$$\begin{aligned} \max & \quad 2x_1 + 3x_2 \\ \text{subject to} & \quad x_1 + 2x_2 \leq 8 \\ & \quad 2x_1 + x_2 \leq 6 \end{aligned}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (1.33, 3.33)

Integer Solution: (0, 4)

Problem # 9 [Salkin, p. 127]

$$\begin{array}{ll} \max & 3x_1 + x_2 \\ \text{subject to} & 2x_1 + 3x_2 \leq 6 \\ & 2x_1 - 3x_2 \leq 3 \end{array}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (2.25, 0.50)

Integer Solution: (1, 1)

Problem # 10 [Ozan, p. 354]

$$\begin{array}{ll} \max & 21x_1 + 27x_2 \\ \text{subject to} & -5x_1 + 15x_2 \leq 30 \\ & 14x_1 + 2x_2 \leq 70 \end{array}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (4.5, 3.5)

Integer Solution: (4, 3)

Problem # 11 [Ozan, p. 352]

$$\begin{array}{ll} \max & 7x_1 + 3x_2 \\ \text{subject to} & -x_1 + 3x_2 \leq 6 \\ & 7x_1 + x_2 \leq 35 \end{array}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (4.5, 3.5)

Integer Solution: (4, 3)

Problem # 12 [Ozan, p. 352]

$$\begin{array}{ll} \max & 4x_1 + 3x_2 \\ \text{subject to} & 4x_1 + x_2 \leq 10 \\ & 2x_1 + 3x_2 \leq 8 \end{array}$$

$$x_1, x_2 \geq 0 \text{ and integer}$$

Continuous Solution: (2.2, 1.2)

Integer Solution: (2, 1)

Problem # 13 [Ozan, p. 352]

$$\begin{aligned} \min & \quad 2x_1 + 3x_2 \\ \text{subject to} & \quad 2x_1 + 7x_2 \geq 12 \\ & \quad x_2 \leq 6 \\ & \quad x_1, x_2 \geq 0 \text{ and integer} \end{aligned}$$

Continuous Solution: (0,1.714)
Integer Solution: (0,2)

Problem # 14 [Ziont, p. 341]

$$\begin{aligned} \max & \quad 5x_1 + 4x_2 \\ \text{subject to} & \quad 3x_1 + 3x_2 \leq 10 \\ & \quad 12x_1 + 6x_2 \leq 24 \\ & \quad x_1, x_2 \geq 0 \text{ and integer} \end{aligned}$$

Continuous Solution: (0.666,2.666)
Integer Solution: (1,2)

Problem # 15 [Hillier and Lieberman, p. 715]

$$\begin{aligned} \max & \quad x_1 + 5x_2 \\ \text{subject to} & \quad x_1 + 10x_2 \leq 20 \\ & \quad x_1 \leq 2 \\ & \quad x_1, x_2 \geq 0 \text{ and integer} \end{aligned}$$

Continuous Solution: (2.0,1.8)
Integer Solution: (0,2)

Problem # 16 [Ozan, p. 351]

$$\begin{aligned} \max & \quad 2x_1 + 3x_2 \\ \text{subject to} & \quad 5x_1 - 2x_2 \leq 28 \\ & \quad x_1 + 2x_2 \leq 35 \\ & \quad x_1, x_2 \geq 0 \text{ and integer} \end{aligned}$$

Continuous Solution: (10.5,12.25)
Integer Solution: (9,13)

Problem # 17 [Salkin, p. 119]

min $2x_1 + 5x_2$
subject to $2x_1 + 2x_2 \geq 9$
 $2x_1 + 6x_2 \geq 22$
 $x_1, x_2 \geq 0$ and integer
Continuous Solution: (1.25, 3.25)
Integer Solution: (2, 3)

Problem # 18 [Taha, 1987, p. 338]

max $x_1 + x_2$
subject to $2x_1 + 5x_2 \leq 16$
 $6x_1 + 5x_2 \leq 30$
 $x_1, x_2 \geq 0$ and integer
Continuous Solution: (3.5, 1.8)
Integer Solutions: (5, 0); (3, 2); (4, 1)

Problem # 19 [Ozan, p. 325]

max $2x_1 + 8x_2$
subject to $2x_1 - 6x_2 \leq 3$
 $-x_1 + 4x_2 \leq 5$
 $2x_1 + 2x_2 \leq 13$
 $x_1, x_2 \geq 0$ and integer
Continuous Solution: (4.2, 2.3)
Integer Solution: (4, 2)

Problem # 20 [Taha, 1975, p. 226]

max $2x_1 + 4x_2$
subject to $2x_1 + 6x_2 \leq 23$
 $x_1 - x_2 \leq 1$
 $x_1 + x_2 \leq 6$
 $x_1, x_2 \geq 0$ and integer
Continuous Solution: (3.25, 2.75)
Integer Solution: (2, 3)

Problem # 21 [Hillier and Lieberman, p. 743]

$$\begin{aligned} \max \quad & 33x_1 + 12x_2 \\ \text{subject to} \quad & -x_1 + 2x_2 \leq 4 \\ & 5x_1 + 2x_2 \leq 16 \\ & 2x_1 - x_2 \leq 4 \end{aligned}$$

$x_1, x_2 \geq 0$ and integer

Continuous Solution: (2.666, 1.333)
Integer Solution: (2, 3)

Problem # 22 [Taha, 1987, p. 836]

$$\begin{aligned} \max \quad & 2x_1 + x_2 + 2x_3 \\ \text{subject to} \quad & 2x_1 + x_2 + x_3 \leq 9 \\ & x_1 + 2x_2 + 3x_3 \leq 8 \end{aligned}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (3.8, 0, 1.4)
Integer Solution: (4, 0, 1)

Problem # 23 [Fogiel, p. 520]

$$\begin{aligned} \max \quad & 9x_1 + 6x_2 + 5x_3 \\ \text{subject to} \quad & 2x_1 + 3x_2 + 7x_3 \leq 17.50 \\ & 4x_1 + 9x_3 \leq 15 \end{aligned}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (3.75, 3.333, 0)
Integer Solution: (3, 3, 0)

Problem # 24 [Fogiel, p. 200]

$$\begin{aligned} \max \quad & 3x_1 + 2x_2 + x_3 \\ \text{subject to} \quad & 2x_1 + 5x_2 + x_3 \leq 12 \\ & 6x_1 + 8x_2 \leq 22 \end{aligned}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (3.666, 0, 4.666)
Integer Solution: (3, 0, 6)

Problem # 25 [Lovett]

$$\begin{array}{ll} \max & x_1 + 2x_2 + 3x_3 \\ \text{subject to} & x_1 + 2x_2 + 3x_3 \leq 10 \\ & x_1 + x_2 \leq 5 \\ & x_1 \leq 1 \end{array}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (1,4,0.333)
Integer Solutions: (0,5,0); (1,3,1)

Problem # 26 [Taha, 1987, p. 337]

$$\begin{array}{ll} \max & 4x_1 + 6x_2 + 2x_3 \\ \text{subject to} & 4x_1 - 4x_2 \leq 5 \\ & -x_1 + 6x_2 \leq 5 \\ & -x_1 + x_2 + x_3 \leq 5 \end{array}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (2.5,1.25,6.25)
Integer Solution: (2,1,6)

Problem # 27 [Hillier and Lieberman, p. 746]

$$\begin{array}{ll} \max & 2.1x_1 + 1.5x_2 + 1.15x_3 \\ \text{subject to} & 33.5x_1 + 25x_2 + 17.5x_3 \leq 750 \\ & x_1 + x_2 + x_3 \leq 30 \\ & .6x_1 + .75x_2 + x_3 \leq 40 \end{array}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (14.0625,0,15.9375)
Integer Solution: (14,0,16)

Problem # 28 [Ozan, p. 352]

$$\begin{array}{ll} \max & 2x_1 + x_2 + 4x_3 \\ \text{subject to} & 2x_1 - 3x_2 + 3x_3 \leq 20 \\ & 12x_1 + 3x_2 - x_3 \leq 10 \\ & 2x_1 + x_2 + x_3 \leq 60 \end{array}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (0,8.333,15)
Integer Solution: (0,8,14)

Problem # 29 [Taha, 1987, p. 337]

$$\begin{aligned} \max \quad & 3x_1 + x_2 + 3x_3 \\ \text{subject to} \quad & -x_1 + 2x_2 + x_3 \leq 4 \\ & 4x_2 - 3x_3 \leq 2 \\ & x_1 - 3x_2 + 2x_3 \leq 3 \end{aligned}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (5.333, 3.0, 3.333)

Integer Solution: (5, 2, 2)

Problem # 30 [Fogiel, p. 258]

$$\begin{aligned} \max \quad & 3x_1 + x_2 + 3x_3 \\ \text{subject to} \quad & 2x_1 + x_2 + x_3 \leq 2 \\ & x_1 + 2x_2 + 3x_3 \leq 5 \\ & 2x_1 + 2x_2 + x_3 \leq 6 \end{aligned}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (0.2, 0, 1.6)

Integer Solution: (0, 0, 1)

Problem # 31 [Hillier and Lieberman, p. 743]

$$\begin{aligned} \max \quad & 4x_1 - 2x_2 + 7x_3 \\ \text{subject to} \quad & x_1 + 5x_3 \leq 10 \\ & x_1 + x_2 - x_3 \leq 1 \\ & 6x_1 - 5x_2 \leq 0 \end{aligned}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (1.25, 1.5, 1.75)

Integer Solution: (0, 0, 2)

Problem # 32 [Salkin, p. 155]

$$\begin{aligned} \max \quad & x_1 - x_2 + 2x_3 \\ \text{subject to} \quad & 2x_1 + 4x_2 - x_3 \leq 20 \\ & -8x_1 + x_2 + 3x_3 \leq 10 \\ & 2x_1 - 9x_2 + 8x_3 \leq 6 \end{aligned}$$

$x_1, x_2, x_3 \geq 0$ and integer

Continuous Solution: (1.9571, 5.6857, 6.6571)

Integer Solution: (4, 4, 4)

Problem # 33 [Salkin, p. 144]

$$\begin{array}{ll} \max & x_1 + x_2 + x_3 \\ \text{subject to} & -4x_1 + 5x_2 + 2x_3 \leq 4 \\ & -2x_1 + 5x_2 \leq 5 \\ & 3x_1 - 2x_2 + 2x_3 \leq 6 \\ & 2x_1 - 5x_2 \leq 1 \end{array}$$

$$x_1, x_2, x_3 \geq 0 \text{ and integer}$$

Continuous Solution: (3.6304, 2.4545, 0)

Integer Solution: (3, 2, 0)

Problem # 34 [Salkin, p. 139]

$$\begin{array}{ll} \max & 4x_1 + 6x_2 + 3x_3 \\ \text{subject to} & x_1 + 2x_2 \leq 5 \\ & 9x_1 + 2x_2 - 4x_3 \leq 8 \\ & -3x_1 - 2x_2 + 2x_3 \leq 1 \\ & -5x_1 + 4x_2 + 6x_3 \leq 16 \end{array}$$

$$x_1, x_2, x_3 \geq 0 \text{ and integer}$$

Continuous Solution: (3.2941, 0, 5.4118)

Integer Solution: (3, 0, 5)

Problem # 35 [Ozan, p. 354]

$$\begin{array}{ll} \max & 10x_1 + 6x_2 + 2x_3 \\ \text{subject to} & 2x_1 + 2x_2 + 2x_3 \leq 12 \\ & 3x_1 + x_2 + 4x_3 \leq 9 \\ & x_1 \leq 1 \\ & x_2 \leq 1 \\ & x_3 \leq 4 \end{array}$$

$$x_1, x_2, x_3 \geq 0 \text{ and integer}$$

Continuous Solution: (1, 1, 1.25)

Integer Solution: (1, 1, 1)

Problem # 36 [Fogiel, p. 83]

$$\begin{aligned} \max \quad & 8x_1 + 15x_2 + 6x_3 + 20x_4 \\ \text{subject to} \quad & x_1 + 3x_2 + x_3 + 2x_4 \leq 9 \\ & 2x_1 + 2x_2 + 2x_3 + 3x_4 \leq 12 \\ & 3x_1 + 3x_2 + 2x_3 + 5x_4 \leq 16 \end{aligned}$$

$$x_1, x_2, x_3, x_4 \geq 0 \text{ and integer}$$

Continuous Solution: (0, 1.444, 0, 2.333)

Integer Solution: (1, 1, 0, 2)

Problem # 37 [Conley, p. 45]

$$\begin{aligned} \max \quad & 4x_1 + 5x_2 + 9x_3 + 5x_4 \\ \text{subject to} \quad & x_1 + 3x_2 + 9x_3 + 6x_4 \leq 16 \\ & 6x_1 + 6x_2 + 7x_4 \leq 19 \\ & 7x_1 + 8x_2 + 18x_3 + 3x_4 \leq 44 \end{aligned}$$

$$x_1, x_2, x_3, x_4 \geq 0 \text{ and integer}$$

Continuous Solution: (2.8652, 0, 1.2871, 0.2584)

Integer Solution: (1, 2, 1, 0)

Problem # 38 [Parker and Rardin, p. 401]

$$\begin{aligned} \min \quad & 10x_1 + 8x_2 + 3x_3 + 11x_4 \\ \text{subject to} \quad & x_1 + x_3 + 7x_4 \geq 10 \\ & 3x_1 + 5x_2 + x_3 + x_4 \geq 5 \\ & 2x_2 + x_3 \geq 2 \end{aligned}$$

$$x_1, x_2, x_3, x_4 \geq 0 \text{ and integer}$$

Continuous Solution: (0, 0.5652, 0.8696, 1.3043)

Integer Solution: (0, 0, 4, 1)

Problem # 39 [Taha, 1987, p. 106]

$$\begin{aligned} \max \quad & 2x_1 + x_2 - 3x_3 + 5x_4 \\ \text{subject to} \quad & x_1 + 7x_2 + 3x_3 + 7x_4 \leq 46 \\ & 3x_1 - x_2 + x_3 + 2x_4 \leq 8 \\ & 2x_1 + 3x_2 - x_3 + x_4 \leq 10 \end{aligned}$$

$$x_1, x_2, x_3, x_4 \geq 0 \text{ and integer}$$

Continuous Solution: (0, 1.7143, 0, 4.8571)

Integer Solution: (0, 2, 0, 4)

Problem # 40 [Taha, 1987, p. 106]

$$\begin{aligned} \max \quad & -2x_1 + 6x_2 + 3x_3 - 2x_4 \\ \text{subject to} \quad & x_1 + 7x_2 + 3x_3 + 7x_4 \leq 46 \\ & 3x_1 - x_2 + x_3 + 2x_4 \leq 8 \\ & 2x_1 + 3x_2 - x_3 + x_4 \leq 10 \end{aligned}$$

$x_1, x_2, x_3, x_4 \geq 0$ and integer

Continuous Solution: (0,2.2,10.2,0)

Integer Solution: (0,4,6,0)

Problem # 41 [Taha, 1987, p. 106]

$$\begin{aligned} \max \quad & 3x_1 - x_2 + 3x_3 + 4x_4 \\ \text{subject to} \quad & x_1 + 7x_2 + 3x_3 + 7x_4 \leq 46 \\ & 3x_1 - x_2 + x_3 + 2x_4 \leq 8 \\ & 2x_1 + 3x_2 - x_3 + x_4 \leq 10 \end{aligned}$$

$x_1, x_2, x_3, x_4 \geq 0$ and integer

Continuous Solution: (0,2.2,10.2,0)

Integer Solution: (0,2,10,0)

Problem # 42 [ISE 326 Project]

$$\begin{aligned} \max \quad & 200x_1 + 150x_2 + 150x_3 + 250x_4 \\ \text{subject to} \quad & 22x_1 + 27x_3 \leq 120 \\ & 5x_1 + 5x_2 + 2x_3 + 4x_4 \leq 200 \\ & 2x_1 + 8x_2 + x_3 + 5x_4 \leq 140 \\ & x_1 + 10x_2 + 8x_4 \leq 110 \\ & 4x_1 + 8x_2 + 16x_3 + 10x_4 \leq 240 \\ & 10x_1 + 14x_2 + 8x_3 + 12x_4 \leq 320 \end{aligned}$$

$x_1, x_2, x_3, x_4 \geq 0$ and integer

Continuous Solution: (5.4545,0,0,13.0682)

Integer Solution: (5,0,0,13)

Problem # 43 [Conley, p. 54]

$$\begin{aligned} \text{max} \quad & 20x_1 + 30x_2 + 40x_3 + 45x_4 + 55x_5 + 60x_6 \\ \text{subject to} \quad & 8x_1 + 8x_2 + 3x_3 + 12x_4 + 8x_5 + 4x_6 \leq 150 \\ & 2x_1 + 4x_2 + 10x_3 + 4x_4 + 2x_5 + 6x_6 \leq 150 \\ & 3x_1 + 1x_2 + 4x_3 + 2x_4 + 4x_5 + 10x_6 \leq 150 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \text{ and integer} \end{aligned}$$

Continuous Solution: (0,0,8.5366,0,12.1951,6.7073)

Integer Solution: (0,0,8,0,12,7)

Problem # 44 [Conley, p. 146]

$$\begin{aligned} \text{max} \quad & 20x_1 + 30x_2 + 35x_3 + 50x_4 + 62x_5 + 66x_6 + 70x_7 \\ & + 81x_8 + 90x_9 + 100x_{10} \\ \text{subject to} \quad & x_1 + 4x_2 + 3x_3 + 2x_4 + 5x_5 + 1x_6 + 2x_7 \\ & + 5x_8 + x_9 + x_{10} \leq 2,700 \\ & 6x_1 + x_2 + 2x_3 + 7x_4 + 3x_5 + 3x_6 + 2x_7 \\ & + x_8 + 6x_9 + x_{10} \leq 2,700 \\ & x_1 + 4x_2 + 3x_3 + 2x_4 + x_5 + 4x_6 + 2x_7 \\ & + 2x_8 + x_9 + 2x_{10} \leq 2,700 \\ & 5x_1 + 5x_2 + 2x_3 + x_4 + 2x_5 + x_6 + 3x_7 \\ & + 3x_8 + x_9 + 5x_{10} \leq 2,700 \end{aligned}$$

$x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10} \geq 0$ and integer

Continuous Solution: (0,0,0,0,0,330.8824,0,397.0588,
185.2941,198.5294)

Integer Solution: (0,0,0,0,0,327,0,397,187,199)

Problem # 45 [Conley, p. 141]

$$\begin{aligned} \text{max} \quad & 20x_1 + 15x_2 + 19x_3 + 27x_4 + 34x_5 + 42x_6 + 58x_7 \\ & + 21x_8 + 90x_9 + 66x_{10} + 15x_{11} + 75x_{12} + 14x_{13} \\ & + 88x_{14} + 62x_{15} + 60x_{16} + 58x_{17} + 54x_{18} + 90x_{19} \\ & + 29x_{20} \\ \text{subject to} \quad & 4x_1 + 5x_2 + 2x_3 + 2x_4 + x_5 + 5x_6 + 6x_7 + 5x_8 \\ & + 4x_9 + 3x_{10} + 5x_{11} + 6x_{12} + 2x_{13} + 8x_{14} + 6x_{15} \\ & + 6x_{16} + 5x_{17} + x_{18} + x_{19} + 5x_{20} \leq 3,800 \\ & x_1 + x_2 + 8x_3 + 6x_4 + 4x_5 + 2x_6 + 3x_7 + 2x_8 \\ & + 4x_9 + 6x_{10} + x_{11} + 2x_{12} + x_{13} + 2x_{14} + 6x_{15} \\ & + x_{16} + 3x_{17} + 4x_{18} + 2x_{19} + 5x_{20} \leq 3,800 \end{aligned}$$

$$\begin{aligned}
& 3x_1 + 2x_2 + x_3 + 2x_4 + x_5 + x_6 + 3x_7 + x_8 \\
& + x_9 + 2x_{10} + 2x_{11} + 2x_{12} + 5x_{13} + x_{14} + x_{15} \\
& + x_{16} + 6x_{17} + x_{18} + 5x_{19} + 2x_{20} \leq 3,800
\end{aligned}$$

$$\begin{aligned}
& 2x_1 + 2x_2 + 2x_3 + 3x_4 + 2x_5 + 2x_6 + x_7 + 4x_8 \\
& + x_9 + x_{10} + 3x_{11} + 2x_{12} + 7x_{13} + 6x_{14} + 2x_{15} \\
& + 1x_{16} + 7x_{17} + 5x_{18} + 3x_{19} + 4x_{20} \leq 3,800
\end{aligned}$$

$x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18}, x_{19}, x_{20} \geq 0$ and integer

Continuous Solution: (0,0,0,0,0,0,0,0,0,575.7576,0,0,0,0,0,0,0,0,345.4545,0,230.3030,0)

B&B Integer Solution: (0,0,0,0,0,0,0,0,0,573,0,0,0,0,0,0,0,0,344,3,232,0)

IESIP Integer Solution: (0,0,0,0,0,0,0,0,0,575,0,0,0,0,0,0,0,0,345,0,231,0)

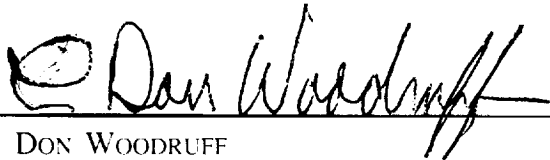
IESIP Integer Solution: (0,0,0,0,0,0,0,0,0,576,0,0,0,0,0,0,0,0,345,0,230,0)

APPROVAL

AN IMPROVED EXPLORATORY SEARCH TECHNIQUE FOR PURE INTEGER LINEAR PROGRAMMING PROBLEMS

By F.R. Fogle

The information in this report has been reviewed for technical content. Review of any information concerning Department of Defense or nuclear energy activities or programs has been made by the MSFC Security Classification Officer. This report, in its entirety, has been determined to be unclassified.



L. DON WOODRUFF
Chief, Systems Analysis Division



for WILLIAM B. CHUBB
Director, Systems Analysis and Integration Laboratory

