THE UNIVERSITY OF MICHIGAN

SYSTEMS ENGINEERING LABORATORY

Department of Electrical Engineering
College of Engineering

SEL Technical Report No. 56

FINDING OPTIMAL GROUPS - A QUADRATIC PROGRAMMING APPROACH IN ZERO/ONE VARIABLES WITH APPLICATIONS

by

Don Michael Coleman

under the direction of
Professor Keki B. Irani

May 1971

under contract with:

ROME AIR DEVELOPMENT CENTER
Research and Technology Division
Griffiss Air Force Base, New York
Contract No. F30602-69-C-0214

enm

UMR0948

Abstract

On Finding Optimal Groups - A Quadratic Programming Approach in Zero/One Variables with Applications

by

Don Michael Coleman

Co-chairman K. B. Irani
E. L. Lawler

In this research we consider a mathematical programming problem of the following form:

$$\text{Maximize} \qquad f(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j - \sum_j b_j x_j$$

$$\text{subject to} \qquad \sum_j a_j^k x_j \leq \mu_k \qquad k=1, 2, \ldots, m,$$

where b_j is an arbitrary real, $c_{ij} \geq 0$ for all i, j, and a_j^k, μ_k are positive reals for all j and k, $x_j \in \{0, 1\}$ for all j.

For values of $m \leq 2$ we present an algorithm for solution of this problem based upon generalized Lagrange multipliers. This algorithm is suitable for application to problems with a rather large number of binary variables.

This formulation is shown to be applicable to a number of optimal grouping problems; a particular application to computer systems paging is examined in detail.

Acknowledgments

I wish to express my gratitude to Professors L. K. Flanigan, B. A. Galler, K. B. Irani, E. L. Lawler, and E. L. McMahon for serving as my doctoral committee, and for their many constructive suggestions and criticisms. I am particularly indebted to Professor K. B. Irani for his professional guidance and direction.

For the task of supervising the typing of this manuscript, I express my thanks to Miss Joyce Doneth.

A special note of thanks is due to Mr. Dayrl Knoblock of The University of Michigan Computing Center staff who provided a special version of the SNOBOL translator and gave invaluable assistance in the collecting of statistics on paging behavior.

This research was supported by the Rome Air Development Center of the United States Air Force under contract no. AF30602-69-C-0214; for this aid I am truly grateful.

Finally, I thank my wife for her patience, understanding, and encouragement which made this goal possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

1. Introduction

In the design of many large scale systems it is frequently useful to formulate a number of the engineering problems as optimization problems in zero/one variables.

Much effort is directed towards developing algorithms for solution of these problems which are readily programmed and do not require unreasonable amounts of computation time and storage.

Linear programming has been one of the more successful techniques developed to deal with problems of this nature. For example, special algorithms have been developed for efficient solution of the classical transportation and assignment problems.

In this dissertation we provide a method for solving a quadratic binary programming problem. We show this problem to have a wide range of applicability in both operations research and in computer systems design.

In the subsequent Sections of this chapter we present first a description of the general quadratic problem with which we will be concerned and then a review of the areas of applicability and known results.

In the concluding portion of this chapter we give the scope of this dissertation in detail.

1.1 General Problem Area

The quadratic programming problem will be defined as follows:

(1) Maximize $f(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j - \sum_j b_j x_j$

subject to

(2) $\sum_j a^k_j x_j \leq \mu_k \qquad k = 1, 2, \ldots, m$

where $\underline{x} = (x_1, x_2, \ldots, x_n)$ $x_i \in \{0,1\}$ for all i, $c_{ij} \geq 0$, for all i and j, a^k_j, μ_k are positive real numbers for all j and k, b_j is arbitrary real number.

For small values of n (less than 40) a general algorithm such as the one given by Lawler and Bell [31] can be used to solve this problem. However, when n becomes relatively large, these algorithms are not efficient when directly applied to this problem.

In this dissertation we present an algorithm although not as general as [31], which can be used to solve (1) and (2) when $m \leq 2$. Our algorithm may be used to solve problems for relatively large values of n without requiring prohibitive amounts of computation time and storage.

In the following section we explore some areas of applicability of problem (1) and (2) for the case of $m \leq 2$.

1.2 Areas of Applicability

There are numerous areas for application of the problem defined in section1.1. In the following three sections we give some operations research problems which may be modelled using the formulation presented in section 1.1.

We begin with a problem we define as the quadratic knapsack problem which is seen to be a special case of the problem of section 1.1.

1.2.1 Quadratic Knapsack Problem

We define the quadratic knapsack problem as follows:

$$\text{Maximize } f(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j$$

subject to

$$\sum_i a_i x_i \leq \mu$$

where $\underline{x} = (x_1, x_2, \ldots, x_n)$, $x_i \in \{0,1\}$ for all i, $\mu > 0$, $c_{ij} \geq 0$ for all i,j, $a_i > 0$ for all i.

Here, as opposed to the classical linear knapsack problem [8], the value associated with selection of a particular item is related to the set of items which are selected along with it. Of course $x_i = 1$ implies that the i^{th} item has been selected while $x_i = 0$ implies rejection. The constant c_{ij} is a measure of the value of selecting items i and j while a_i is the weight of the i^{th} item and μ is a constraint on the total weight of the items selected.

A particular instance where such a model is applicable is in the program planning and budgeting field [2] ,[5] , [45].

1.2.2 Optimal Selection of R and D Projects

It can be seen from the literature that the problem of selecting a firm's research projects has been given considerable attention (see

[5] for example). In most of the work previously published, considerable attention has been focused on the research and development allocation problems which can be formulated as linear programming models. An interesting variation of the optimal project selection problem is handled by the quadratic knapsack problem.

The problem briefly stated is as follows. A large research and development concern is faced with the problem of selecting from among n projects those which should be researched. It is further assumed that among the n projects one has quantified a matrix $\{c'_{ij}\}$ which represents the value of success,

c'_{ij} = the value of the successful outcome of the i^{th} and j^{th} research proposal

One might see [2] for a discussion of how one estimates the value of successful projects. There also might be associated with each pair of projects a number p_{ij} the probability of a successful outcome of the i^{th} and j^{th} project. Therefore we can define $c_{ij} = p_{ij}\, c'_{ij}$ which represents a weighted interrelated value coefficient. And we select the set of projects which maximize

$$(1) \qquad \sum_{i,j} c_{ij} x_i x_j \, .$$

Of course $x_i = 1$ implies that the i^{th} project has been selected while $x_i = 0$ implies otherwise. Now usually management estimates, from past experience, the cost per man-hour associated with a given type of research project. This is represented by a_i for the i^{th} project.

The total available man-hours per year μ for all projects which are utilized among the projects is a cost constraint known by management. And it is this constraint which must be satisfied when the set of research projects are selected i.e.,

$$\sum_i a_i x_i \leq \mu. \tag{2}$$

The maximization of (1) subject to (2) gives the best selection of research projects and is a direct application of quadratic knapsack problem.

There are numerous algorithms in the literature for the classical knapsack problem [8], [9], [17]. However there are few results concerning the quadratic case. In chapter four we give some computational results for the problem of selecting an optimal set with inter-related value coefficients.

The next section illustrates an application in the area of pattern recognition.

1.2.3 Pattern Recognition

A common problem in pattern recognition [43] is that of classifying signals into sets of common patterns, i.e., signals or points which are somehow similar are expected to be classified into different classes.

Essentially there are three basic elements to the classification problem in pattern recognition. The first is that of metricizing, i.e., a method of measuring distance between the points to be classified.

Generally the distance measure is a way of quantifying the similarity or dissimilarity of the points to be classified. Next is the concept of a clustering criteria which usually includes the notion that "distances" between members of the same class are on the average small. Finally, there must be a computational procedure for selecting the clusters or partitioning the set of points.

If the first two elements of the pattern classification problem are given, our quadratic programming formulation can be used as a method for classification. For example let us have a set of points $\{1,2,\ldots,n\}$ for which we have a metric defined on all pairs of points. If c'_{ij} = the distance between the i^{th} and j^{th} point define $c_{ij} = \frac{1}{c'_{ij}}$. It is desired to partition the set of points into two sets such that the sum of the distances of the points in the same set is minimized. One could select this partition by solving the following:

$$\text{maximize} \quad \sum_{i,j} c_{ij} x_i x_j + \sum_{i,j} c_{ij}(1-x_i)(1-x_j)$$

subject to

$$\sum_i x_i \geq \mu_1$$

$$\sum_i x_i \leq \mu_2$$

$$x_i \in \{0,1\} \text{ for all } i, \quad \mu_1 < \mu_2 .$$

This is a special case of the problem defined in section 1.1. The numbers μ_1 and μ_2 could be used to control the size of the partition, i.e., how many elements in each set of partitions. The first term in

the objective function is a measure of the similarity of the points which are selected for first set of the partitions while the seond term measures the similarity of those points which are not selected and which form the second set of the partition.

In the next section we discuss a problem in computer storage allocation. The study of this problem was a motivation for looking at problems of optimal groupings and hence our quadratic problem.

1.2.4 A Problem in Computer Systems Storage Allocation

In the relatively short time that the electronic digital computer has been available, it has undergone extensive changes in its basic make-up and organization. This transformation has seen the computer move from a relatively slow, unreliable machine with limited memory capacity to a highly complex system of processors with a hierarchy of memories and complicated input-output devices. Along with the emergence of new hardware has been the emergence of the art of multiprogramming. This concept has been aided by development of the virtual storage machine [1] , i.e., each user in the system may assume that he has available a very large address space, which may be many times larger than the size of main memory. The bookkeeping and memory management for efficient operation of such a system is automatic and transparent to the user.

Multiprogramming, as the name implies, means having several programs occupy high speed memory simultaneously. The problems

of effectively using storage in a multiprogrammed mode are sometimes appropriately called problems of storage allocation. When we consider this problem in the context of the early days of computing it was relatively straightforward.

In early batch systems where programs were run one at a time, each program had the entire high speed memory (core) available. Problems arose when a program was larger than the available core. In these cases the programmer had to improvise by "segmenting" his program (instructions and data), and controlling the overlaying of "segments". Segmenting, of course, referred to dividing the program up into parts (segments); hence during execution the unused segments of a program were kept on a backing store and brought into core, i.e., overlaid as they were needed. If the problem of segmenting a program was given to the operating system instead of the programmer we had what was called an automatic segmenting system; and the problem of segmenting the program in an optimum manner became known as the classical overlay problem [36] or the problem of program segmentation [26]. Segmentation has come to mean something slightly different in modern terminology [11].

In multiprogrammed systems overlaying or segmentation is intended to increase the size of effective memory. Segmentation can also reduce the delay in loading a large task into the space currently occupied by many smaller ones, since in the case of segmented systems

much of the overhead incurred by having to move tasks around to make the larger task fit can be avoided. Likewise the delay is reduced in loading small tasks among large tasks. Therefore with effective segmentation throughput may be increased. A summary of techniques for overlaying is found in [36].

1.2.4.1 Paging

In batch systems there was no particular problem associated with the overlays of different sizes (the only constraint was that they were smaller than the core). However in multiprogramming systems, if the size of the unit of core allocation is varied to suit the needs of the information being transferred, then the problem of storage fragmentation is incurred; that is, the high speed memory available for storage becomes fragmented into numerous little sets of contiguous locations. There is a high system overhead incurred by physically moving tasks in and about to utilize a fragmented core, due to the added burden of keeping track of things once they have been committed to high speed memory. In the first dynamic storage allocation system, the Ferranti ATLAS, this problem was "solved" as follows. High speed memory was divided into fixed size blocks called page frames. The programs and data which were to be brought into high speed storage were divided into similarly fixed sized blocks called pages. Now when information was to be brought into core, the set of available frames was matched with the set of blocks or pages which were to be brought in. Systems which

use this approach usually have hardware mapping devices which make the address of items in a page independent of the particular page frame in which a given item resides. Such systems are called paging systems and these are the systems with which we will be directly concerned. Actually the physical dividing up of core does not "solve" the core fragmentation problem. However, in such systems fragmentation is now within individual pages rather than in high speed memory. This is because it is often the case that a page(s) allocated to a task will only be partially used. The great virtue of such systems is that they are simpler to implement since a page can be placed into any available page frame. Some modern systems do not have a uniform unit of core allocations, for example the MULTICS system has two such frame sizes - 1024 words and 64 words, and are still referred to as paging systems. A good summary of the allocation strategies and hardware facilities of several modern day systems is given in [41].

A basic problem in paging systems is that of memory management, that is, the decisions by which the operating system determines what pages are to be brought in to high speed memory, called fetch strategies, the decisions which determine where the incoming pages are to be placed, called placement strategies, and the decisions to determine which pages must be removed when the conditions of high speed core so dictate, called replacement strategies. A set of such strategies is usually called a page turning rule or algorithm. These strategies must

be formulated in the total system environment with consideration given to such system policies as scheduling, which is most often concerned with the selection of the task to use the CPU, sharing of programs, and other system features. However, the main objective of such page-turning policies is to decrease the page traffic between high speed memory and the backing store. The reasons for such an objective are well known. First, there is a system's overhead associated with making a decision about what to move and where to move it. Second if the volume of transfer between the two levels of storage grows too fast, there can be congestion in the channel between the two memories which could in some cases seriously affect processor efficiency due to tasks being queued for the use of the channel. The operation of the CPU is further degraded, by the "stealing" of memory cycles by other auxiliary memory devices, if the page traffic is high.

In most of the literature the memory management problem is divided into two stages:

(1) Paging in - Locate the required page in the backing store, bring it into the high speed memory and turn on the appropriate bit in the appropriate page table.

(2) Paging out - Remove some page from main memory; turn the in-core bit off in the appropriate page table.

Denning [10] notes that the problem of paging in is solved apriori since most systems use the so called demand paging strategy, a page is

brought in only when it is first referenced by an executing task. So most work is concerned with the page replacement part of the management strategy. A summary of some of the current replacement rules is given in [10].

One current view is that by careful analysis of the structural and behavioral characteristics of programs which run on paged systems, one can decrease the page traffic. Structural and behavioral characteristics of a program are referred to as microscopic properties as opposed to its macroscopic properties, e.g., execution time, size, etc. The structural characteristics are determined by the threads of control between blocks of program code while behavioral characteristics refer to frequencies of references between program blocks.

In Chapter 4 we present a detailed discussion of models for program reorganization. In this chapter we show how the quadratic programming problem may be used as a tool for selecting a block of a computer program.

1.2.4.2 Class of Computer Programs for Partitioning

Given the prevailing conditions for collecting statistics on program behavior, we initially consider program reorganizational techniques to apply to a single class of programs. These are large systems programs which are run frequently, such as compilers, assemblers, and certain library routines. These programs tend to be composed of a large number of subroutines and tend to be combined at execution

according to a rather constant pattern. Such programs are heavily used and are available for study. The time spent gathering statistics for model building is justified if only a small increase in operating efficiency is obtained.

In our subsequent discussions we draw freely on the descriptive terminology of the graph theory. For the reader unfamiliar with such terminology we refer him to Appendix III.

1.2.4.3 A Review of Program Partitioning

We now review what has been set forth in the past regarding the microscopic approach to the problem of memory management.

Ramamoorthy [39] has previously proposed an operating system which would detect, during the compilation phase, those structural properties of a program which would be incorporated into a page-turning algorithm. He advocates an advance paging policy as opposed to demand paging. There is some merit to advance rather than demand paging. Although demand paging tends to minimize the amount of high speed memory allocated to the tasks in the system, since only pages which are referenced are loaded, a more significant measure of a strategy's effectiveness is the space time product. A program awaiting pages will continue to occupy high speed memory; in this case the space time product will be affected by the time taken to fetch pages. Pinkerton [38] has results which show conditions under which an advance paging policy is more effective, in terms of space time product, than demand

paging. More important to our immediate goals are the suggestions Ramamoorthy has about how pages should be produced, what we call the "page packing problem". He advocates that the blocks of code placed on the pages (packing of pages) after the program has been compiled should be chosen on the basis of connectivity and frequency of access. He uses the directed graph model of a program, as previously introduced by Karp [22] and Marimont [34], as the basic tool for study of the page packing problem. Each node of the graph represents a small block of data or instructions. The directed arcs represent the transfer of control between instructions, and the undirected edges are used to indicate an instruction referencing data. Since the edge is undirected control is to remain in the instruction block.

Ramamoorthy uses the connection matrix of the graph to determine a number of useful program properties. For example, the program can be examined for logical consistency, i.e., no infinite loops, or redundant instructions by a series of algebraic operations on the connection matrix. His main result concerning the packing of programs has to do with the observation of the property of strong connectedness among the nodes of the graph. A set of nodes are strongly connected if given any pair of nodes v_i, v_j there is a path from v_i to v_j and also a path from v_j to v_i. Ramamoorthy states that the page packing which minimizes traffic between high speed memory and the backing store will be one which places all nodes of the strongly connected components of a graph on the same page. However, since the strongly connected

components of a graph can be any size, ranging from only one instruction to perhaps the complete program, the fact that there are constraints on the page sizes indicates that this result will in general not be very useful for practical programs.

Kral [26] has formulated the problem of program segmentation as a problem in pseudo-boolean programming. His paper consists of showing that the packing of a program can be reduced to the problem of solving a linear integer programming problem in zero-one variables. He concludes by observing that all the algorithms of pseudo-boolean programming can be applied to this problem. However, most of the known algorithms are not sufficiently effective to solve problems of the ranges involved in program segmentation. The essential part of this paper is the formulation of the linear integer programming problem. In this paper Kral gives no algorithm for the problem solution.

Pinkerton [38] uses a Markov model for study of the problem of selecting the segments of a program. An optimization problem is formulated. No algorithm is given for the solution of the optimization problem. He concludes that the cost of an effective solution to the problem is not justified in terms of all the assumptions necessary in the building of the model.

Kernighan [24] studies the problem of assigning subroutines to pages and graphical partitioning. He develops an algorithm for assignment of subroutines to pages such that interpage transfers are minimized under the following conditions.

1. The program graph is a tree
2. The edge weights of the graph are monotone non-decreasing on any path away from the root (initial vertex) of the tree. That is, if v_i, v_j, and v_k are a sequence of vertices on a path away from the root then $c(i,j) \leq c(j,k)$, where $c(i,j)$ is the weight of edge (v_i, v_j).
3. Cost on all edges leaving a vertex is the same. That is if vertex v_i is the initial vertex for arcs $(v_i, v_j), (v_i, v_k), \ldots (v_i, v_\ell)$, then $c(i,j) = c(i,k) = \ldots = c(i, \ell)$.

From this basic procedure an algorithm for the more general case, acyclic graphs, is developed. Under similar restrictions on the edge weights of the graph and with duplication of vertices allowed, Kernighan gives a procedure for optimum assignment of subroutines to pages.

The restrictions of his procedures to acyclic graphs is limiting if the procedure is to be applied to real programs. However, if a suitable methods were available for representing general directed graphs by acyclic graphs, then his algorithms would be a good initial step for program partitioning. Estrin and Martin [13] have given procedures for transformation of certain cyclic structures to acyclic graphs.

Kernighan and Lin [25] have given a heuristic procedure for graphical partitioning. In this case there is no restriction on the type of graph or on the edge weights. Essentially a "hill climbing" algorithm is used to find an optimum partition of a graph into two sets of equal

vertices. The procedure is reapplied to the two subgraphs defined by the two sets of vertices. The procedure is applied as often as necessary to the resulting subgraphs until the required partition is found.

Essentially, the idea of "hill climbing" is to start with an arbitrary two-block partition and move progressively to better two-block partitions until an optimum two-block partition is found. Better two-block partitions are found by interchanging vertices between the blocks of a current partition; a gain g_i is computed for each interchange. The strategy is to maximize the sum $\sum_{i=1}^{k} g_i$ over k interchanges. When this is done, the resulting partition is treated as the initial partition and the procedure is repeated. When the maximum of the sum $\sum g_i$ is zero, a local maximum is reached. Kernighan and Lin give ways of trying to improve this two-way partition. The procedure requires all vertices to be the same size and ways of dealing with unequal size vertices are given.

When the graph is small an algorithm developed at Informatics [20] can be used to find an optimum partition for arbitrary graphs. This algorithm, backtrack programming, is essentially a method of implicit enumeration. A very simple heuristic algorithm is reported in [20]; in Chaper 5 we compare computer results using this algorithm.

In the next section we note a final area of application for our quadratic programming problem.

1.2.5 Other Relevant Results

A great deal of research literature has been devoted to problems of finding optimum groupings. The quadratic programming problem is closely related to many of these, such as the assignment problem, graphical partitioning, and cluster analysis. In the particular area of numerical taxonomy [46] and cluster analysis [21], [42], [44] we find a number of algorithms designed to give groupings of items according to some specified criteria. With cluster analysis the problem is to orgainize a set of data into homogeneous groups where such homogeneity is based upon some measure of associativity between items of data. We note that this problem is significant only when there is a rather large number of elements to be organized, this being on the order of a few hundred. With cluster analysis, there is a measure (objective function) to judge the efficacy of a given partition or grouping of the data. Much effort has been devoted to the problem of extremizing some function of the distance metric [21] which is defined on the set of objects to be partitioned.

When the distance metric is given, the criterion for optimality will depend upon the given application. Typically one is concerned with extremizing the within group distances of the data objects to be grouped or clustered.

For example let c_{ij} be the metric distance between data items i and j of a set of items to be clustered into M groups. Suppose that the criterion is to maximize the within group distances. One common

way of proceeding (for example see [12]) is to divide the data objects into the two most compact groups and repeat the procedure sequentially until M groups are obtained.

Finding the two most compact groups when maximizing the within group distance can be accomplished by solving the following.

$$\text{Maximize} \quad \sum_{i,j} c_{ij} x_i x_j + \sum_{i,j} c_{ij}(1-x_i)(1-x_j)$$

Subject to

$$\sum_i x_i \geq 1$$

$$x_i \in \{0,1\} \quad \text{for all } i$$

$$x_i = 1 \qquad \text{data item } i \text{ in group } 1,$$

$$= 0 \qquad \text{data item } i \text{ in group } 2 .$$

When $M = 2$ the solution to this problem gives the optimal clustering. However when this procedure is used sequentially, i.e., $M > 2$ the solution may very well be suboptimal. However the problem of dividing the partition into two groups where we have only the single constraint $\sum x_i \geq 1$, is solved efficiently by the method of generalized multipliers and our algorithm for unconstrained optimization which we present in chapters two and three.

In reviewing the literature for problems of nontrivial dimension we find no algorithms which guarantee optimality other than total enumeration. Therefore most clustering algorithms are concerned with a search for the best alternative among a small subset of alternatives. Rubin [42] reported an algorithm which gave "good" results for a

clustering problem consisting of 150 items. The algorithm can be classified essentially as a hill climbing procedure, as with Kernighan and Lin, which ran moderately fast for these types of problems; the reported times for this algorithm ranged from 3 to 12 minutes on the IBM 7094.

To obtain a feel for the combinational complexity of problems of this type consider the following. Given a set of N objects, it can be partitioned into M disjoint subsets in S(N,M) ways where S(N,M) is the Stirlings number of the second kind,

$$S(N,M) = \frac{1}{M} \sum_{K=0}^{K=M} (-1)^{M-K} \binom{M}{K} K^N$$

For the following values of M and N Jensen [21] gives the following,

$$S(25,10) = 1,203,163,392,175,387,500$$

$$S(24,6) = 6,090,236,036,084,530$$

So even with program partitioning, where some partitions are infeasible because of size constraints, we are confronted by an enormous combinatorial problem when we consider partitions for sets where $N \approx 125$, $M \approx 20$, as is the case for a typical compiler.

Lawler et al [32] reports an algorithm for partitioning logical networks into modules such that intermodular delay is minimized. The procedures are developed first for networks represented by trees and then generalized for the case of acyclic networks; optimal clusterings are obtained provided duplication is allowed of network elements. The procedure consists of an efficient labelling procedure which is similar to that found in [24].

1.3 Scope of Dissertation

In chapter two we develop a two phase algorithm for maximization of an unconstrained quadratic function. The third chapter is devoted to techniques used to convert constrained optimization problems to unconstrained problems. Chapter four is a chapter of applications in which we give a model for program partitioning along with an algorithm for partitioning based upon solutions of the problem defined in section 1.1. In chapter five we give some alternative strategies for program partitioning and some comparative results. Chapter six is an actual application of program partitioning to a large systems program. Finally in chapter seven we give conclusions and future research problems.

Chapter 2

2. Unconstrained Optimization of a Discrete Quadratic Function

In this chapter we develop an algorithm for solution of an unconstrained quadratic function in zero/one variables. The algorithm will be the basic tool used in solution of the constrained problems defined in the first chapter.

Basically the unconstrained problem is defined as:

$$\text{Maximize } Y(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j - \sum_j b_j x_j \qquad (2.1)$$

where we have $c_{ij} \geq 0$, for all i, j, b_j real and $x_j \in \{0, 1\}$ for all j.

In all our subsequent discussions we assume that $c_{ii} = 0$; since if a term $c_{ii} x_i^2$ is in the objective function it can be replaced by $c_{ii} x_i$. We simply set such a nonzero coefficient c_{ii} to zero and change the linear coefficient of x_i to $b_i - c_{ii}$.

One of the main difficulties in the solution of (2.1) i.e., obtaining a free maximum, is the dimension of the problems with which we are concerned. For solutions of problems with more than about 40 variables, conventional mathematical programming methods do not exist or are inefficient when directly applied. In this chapter we devise an algorithm for obtaining a free maximum which has two phases. The first phase of the algorithm is a direct approach; sufficient conditions are developed to determine if the variables of a given set are either zero or one in a maximizing solution. These tests are not exhaustive, i.e., not all values for all x_i are found but they are sufficiently powerful to find a large number of the boolean values in a maximizing solution. Some computation

results are given using these tests where the test problems ranged from 6 to 150 variables and there were up to 600 nonzero coefficients c_{ij}. The results of these problems where c_{ij} and b_j were randomly generated suggested that the first phase of this algorithm was sufficiently powerful to determine enough of the variables so that those remaining variables in a maximizing solution could be determined by more conventional techniques.

In the second phase of the algorithm we deal with a problem of exactly the same form as (2.1) but of considerably smaller dimension. In this phase we use an implicit enumeration scheme for determination of the remaining variables.

2.1 Phase I Branching Algorithm

In this section we give the proofs of a number of sufficiency tests which allow easy determination of a subset of the variables which maximize (2.1). The first phase of our algorithm is really a structured search of the space of feasible solutions (all boolean n vectors) where we can determine easily which paths are not to be taken if we are to determine an optimum solution vector $\underline{x}^*$.

We start with a test to determine which variables are to be zero for a solution vector $\underline{x}^*$ which maximizes Y in equation (2.1). Note that a lower bound of Y in a free maximum is zero. This observation leads to the following theorem:

Theorem 1a

A boolean vector $\underline{x}^* = (x_1^*, \ldots x_n^*)$ maximizes

$$Y = \sum_{i,j} c_{ij}x_ix_j - \sum_j b_jx_j,\ c_{ij} \geq 0,\ b_j \geq 0 \text{ for all } i,j,$$

only if $x_i^* = 0$ whenever $\sum_j (c_{ij} + c_{ji}) < b_i$.

Proof

Assume that $\underline{x}^* = (x^*_1, \ldots, x^*_{i-1}, 1, x^*_{i+1}, \ldots, x^*_n)$ maximizes Y and suppose that $\sum_j (c_{ij} + c_{ji}) < b_i$. Let us consider $Y(\underline{x}^*) = z + x_i^* (\sum_j (c_{ij} + c_{ji})x_j^* - b_i)$ where z contains all terms of $Y(\underline{x}^*)$ not involving x_i^*. Now since $\sum_j (c_{ij} + c_{ji}) - b_i < 0$ $Y(\underline{x}^*) = x + 1. \times (\sum_j (c_{ij} + c_{ji})x_j^* - b_i) < z + 0$, hence $\underline{x}^*$ cannot maximize Y.

Q.E.D.

Theorem 1b

A boolean vector $\underline{x}^* = (x_1^*, \ldots, x_n^*)$ maximizes

$$Y = \sum_{i,j} c_{ij}x_ix_j - \sum_j b_jx_j, \quad c_{ij} \geq 0, \text{ for all } i,j$$

only if $x_i^* = 1$ whenever $b_i \leq 0$.

Proof

Assume $\underline{x}^* = (x_1^*, \ldots, x^*_{i-1}, 0, x^*_{i+1}, \ldots, x^*_n)$ maximizes Y and suppose that $b_i \leq 0$. Consider $Y(\underline{x}^*) = z + x_i^* \left[\sum_j (c_{ij} + c_{ji})x^*_j - b_i \right]$ where z contains all terms of $Y(\underline{x}^*)$ not involving x^*_i. Now since $b_i \leq 0$ we have $Y(\underline{x}^*) = z + 0 * [\sum_j (c_{ij} + c_{ji})x^*_j - b_i] \leq z + 1 * [\sum_j (c_{ij} + c_{ji})x^*_j - b_i]$.

Q.E.D.

We use these initial theorems as a first step (see flow chart of figure 2.1) in our algorithm for finding a maximizing vector of (2.1).

We next consider additional sufficient conditions for setting a Boolean variable, x_i, of equation (2.1) equal to one.

Sufficient Conditions for $x_i = 1$

Given an x_i we want to decide if it is desirable to set x_i equal to one. The main idea is when we set an x_i equal to one, we should be certain that the contribution to the objective function is nonnegative; since we can always make a nonnegative (zero) contribution, with respect to a given variable, by setting that variable equal to zero in equation (2.1). For example, suppose for some i and j, $c_{ij} \geq b_i + b_j$ in (2.1). Then we know that we can set $x_i=1$ and $x_j=1$, since in this case the contribution to (2.1) can never be negative. We express this sufficient condition more generally in Theorem 2.

Definitions

1) We now define the following "coefficient sets" S_i and S_i^+ which will be used in construction of the algorithm for maximization of Y as

$$S_i = \{j:\ c_{ij} \neq 0 \text{ or } c_{ji} \neq 0\}$$

$$S_i^+ = \{j:\ c_{ij} - b_j \geq 0 \text{ or } c_{ji} - b_j \geq 0\} \subseteq S_i$$

We note that if c_{ij} or c_{ji} and b_j are all zero then $j \notin S_i^+$; and for every i, $i \notin S_i^+$, since $c_{ii} = 0$.

2) We define the assignment sets A, A'_i, and A_i^0 as

$$A = \{j: \text{ optimum value of } x_j \text{ is not known}\}$$

$$A'_i = \{j: c_{ij} \neq 0 \text{ or } c_{ji} \neq 0,\ x^*_j = 1\} \not\subseteq A$$

$$A_i^0 = \{j: c_{ij} \neq 0 \text{ or } c_{ji} \neq 0,\ x^*_j = 0\} \not\subseteq A$$

We note at this point that after application of lemma 1a, we have the following conditions satisfied for each x_i not set to zero and $i \in A$; $\sum_{j \in A} (c_{ij} + c_{ji}) \geq b_i$ and theorem 1b guarantees that $b_i > 0$ if $i \in A$.

Theorem 2

If $\sum_{j \in S_i^+} (c_{ij} + c_{ji}) \geq b_i + \sum_{j \in S_i^+} b_j$, then
$x_t^* = 1$ for all $t \in S_i^+$ and $x_i^* = 1$.

Proof

Let $S_i^+ = \{t_1, t_2, \ldots t_m\}$. Notice $|S_i^+| = m$. We can write the objective function (2.1) as

$$Y(\underline{x}) = \sum_{\ell, j} c_\ell x_\ell x_j - \sum_j b_j x_j = \sum_{j \notin S_i^+} (c_{ij} + c_{ji}) x_i x_j + \sum_{\substack{\ell \in S_i^+ \\ h \in S_i^+}} (c_{\ell h} + c_{h\ell}) x_\ell x_h$$

$$+ \sum_{\substack{j \notin S_i^+ \\ j \neq i}} (c_{t_1} + c_{jt_1}) x_{t_1} x_j + \ldots + \sum_{\substack{j \neq i \\ j \notin S_i^+}} (c_{t_m j} + c_{jt_m}) x_{t_m} x_j + \sum_{\substack{\ell \notin S_i^+ \\ h \notin S_i^+ \\ \ell, h \neq i}} (c_{\ell h} + c_{h\ell}) x_\ell x_h$$

$$+ \sum_{j \in S_i^+} (c_{ij} + c_{ji}) x_i x_j - b_i x_i - \sum_{j \in S_i^+} b_j x_j - \sum_{j \notin S_i^+ \cup \{i\}} b_j$$

Let $I \subset S_i^+ \cup \{i\}$ and $Y^0 \triangleq Y(\underline{x}')$ where,

$x_j' = 0$ for $j \in I$;

$x_j' = 1$ for $j \in [S_i^+ \cup \{i\}] - I$;

x_j' arbitrary for $j \notin S_i^+ \cup \{i\}$.

Let $Y^1 \triangleq Y(\underline{x}'')$ where

$$x_j'' = 1 \qquad \text{for } j \in S_i^+ \cup \{i\} \ ;$$

$$x_j'' = x_j' \qquad \text{for } j \notin S_i^+ \cup \{i\} \ ;$$

We claim $Y^1 - Y^0 \geq 0$.

First let $I = S_i^+ \cup \{i\}$. Then we have

$$Y^1 - Y^0 = \Bigg[\sum_{j \notin S_i^+} (c_{ij} + c_{ji}) x_j + \sum_{\substack{j \neq i \\ j \notin S_i^+}} (c_{t_1 j} + c_{j t_1}) x_j +$$

$$\cdots + \sum_{\substack{j \neq i \\ j \notin S_i^+}} (c_{t_m j} + c_{j t_m}) x_j \Bigg]$$

$$+ \sum_{\substack{\ell \in S_i^+ \\ h \in S_i^+}} c_{\ell h} + \sum_{j \in S_i^+} (c_{ij} + c_{ji}) - b_i - \sum_{j \in S_i^+} b_j \geq 0.$$

The bracketed terms always sum to a nonnegative value, regardless of the values of x_j's. By our hypothesis the sum of the terms outside the brackets is nonnegative; hence we get that any maximizing vector $\underline{x}^*$ cannot have all $x_j^* = 0$, for $j \in S_i^+ \cup \{i\}$. We now show that no proper subset of variables with indices in $S_i^+ \cup \{i\}$, can be zero in a maximizing vector $\underline{x}^*$.

Suppose I is a proper subset of $S_i^+ \cup \{i\}$. Let us assume that $I = \{t_1, t_2 \ldots t_r\}, r \leq m$. Note that in this case $i \notin I$. Then we have

$$Y^1 - Y^0 = \Bigg[\sum_{\substack{j \neq i \\ j \notin S_i^+}} (c_{t_1 j} + c_{j t_1}) x_1 + \ldots + \sum_{\substack{j \neq i \\ j \notin S_i^+}} (c_{t_r j} + c_{j t_r}) x_j \Bigg]$$

$$+ \sum_{\substack{\ell \in I \\ h \in S_i^+}} (c_{\ell h} + c_{h\ell}) + \sum_{t_j \in I} (c_{it_j} + c_{t_j i}) - \sum_{t_j \in I} b_{t_j} .$$

Since for $t_j \in I \subset S_i^+$

$$c_{it_j} - b_{t_j} \geq 0 \qquad \text{or} \qquad c_{t_j i} - b_{t_j} \geq 0$$

and since the terms in brackets are always nonnegative, $Y^1 - Y^0 \geq 0$.

Suppose the set I contains i, that is $I = \{i, t_1, t_2, \ldots, t_r\}$, then

$$Y^1 - Y^0 = \Big[\sum_{j \notin S_i^+} (c_{ij} + c_{ji}) x_j + \sum_{\substack{j \neq i \\ j \notin S_i^+}} (c_{t_1 j} + c_{j t_1}) x_j + \ldots$$

$$\ldots + \sum_{\substack{j \neq i \\ j \notin S_i^+}} (c_{t_r j} + c_{j t_r}) x_j \Big] + \sum_{\substack{\ell \in I \\ \ell \neq i \\ j \in S_i^+}} (c_{\ell j} + c_{j\ell})$$

$$+ \sum_{j \in S_i^+} (c_{ij} + c_{ji}) - b_i - \sum_{t_j \in I} b_{t_j} .$$

Now we know that the bracketed terms have a nonnegative sum and

$$\sum_{j \in S_i^+} (c_{ij} + c_{ji}) - b_i - \sum_{t_j \in I} b_{t_j} \geq \sum_{j \in S_i^+} (c_{ij} + c_{ji}) - b_i - \sum_{j \in S_i^+} b_j \geq 0.$$

Therefore $Y' - Y^0 \geq 0$.

Hence we see that no subset of variables of $\underline{x}$ with indices in $S_i^+ \cup \{i\}$ can be zero for any $\underline{x}$ which maximizes Y. This completes the proof of theorem 2.

The motivation for our next lemma results from examination of the set of assigned variables A_j', for every unassigned variable x_j, $j \in A$. Suppose we have unassigned variables x_j and there exists a

c_{qj} such that $c_{qj} \geq b_j$ and $x^*_q = 1$, i.e., $q \in A_j'$. Then we see that x_j can be assigned the value one, since we are assured that we can get only a nonnegative contribution to Y, our objective function. Writing this condition in a more general form gives theorem 3.

Theorem 3

If $\sum_{j \in A'_i} (c_{ij} + c_{ji}) \geq b_i$, then $x_i^* = 1$.

Proof: Obvious.

The next theorem is a generalization of the theorem 2, in that we consider several sets S_i^+ simultaneously.

Theorem 4

Let $L \subset A$, and let $P = \bigcup_{i \in L} S_i^+$; and suppose $L \subset P$.
If for each $i \in L$ there exists j and k in L, $j \neq k \neq i$, such that $i \in S_j^+$ and $j \in S_k^+$, then $x_\ell^* = 1$ for all $\ell \in P$.

Proof

Let $I \subset P$ and let $I_1 = I \cap L$, $I_2 = I - I_1$. Let $Y^1 \stackrel{\Delta}{=} Y(\underline{x}')$ where

$$x_i' = 1 \quad i \in P$$

$$x_i' \text{ arbitrary } i \notin P.$$

Let $Y^0 \stackrel{\Delta}{=} Y(\underline{x}'')$ where

$$x_i'' = 0 \quad i \in I$$

$$x_i'' = 1 \quad i \in P - I$$

$$x_i'' = x_i' \quad i \notin P$$

We claim $Y^1 - Y^0 \geq 0$ for all $I \subset P$.

$$Y^1 - Y^0 = \sum_{\substack{i\in I \\ j\notin P}} c_{ij}\, x_j + \sum_{\substack{i\in I \\ j\notin P}} c_{ji}\, x_j + \left[\sum_{\substack{i\in I \\ j\in P}} c_{ij} + \sum_{\substack{i\in I \\ j\in P}} c_{ji} - \sum_{i\in I} b_i \right].$$

The terms outside the brackets are nonnegative. We can write the terms within the brackets as

$$\left[\sum_{\substack{i\in I_1 \\ j\in P}} c_{ij} + \sum_{\substack{i\in I_1 \\ j\in P}} c_{ji} - \sum_{i\in I_1} b_i \right] + \left[\sum_{\substack{i\in I_2 \\ j\in P}} c_{ij} + \sum_{\substack{i\in I_2 \\ j\in P}} c_{ji} - \sum_{i\in I_2} b_i \right]$$

Now for each $i \in I_1$, since $I_1 \subset L \subset P$, there exists a $j \in L$ such that $i \in S_j^+$, therefore, we have either $c_{ij} \geq b_i$ or $c_{ji} \geq b_i$. By hypothesis we have that there exists a $k \in L$, where $k \neq i$ such that $j \in S_k^+$. We then have a distinct $c_{jk} \geq b_j$ or $c_{kj} \geq b_j$. Therefore, for any i and j in I_1 we have b_i and b_j "covered" by distinct c's. This means the first of the bracketed expressions is nonnegative.

Suppose $i \in I_2$, then there exists a j such that $j \in L$, $i \in S_j^+$. Now $j \notin I_2$ (by construction), therefore, for each $i \in I_2$ there is distinct c_{ij} or c_{ji} such that either $c_{ij} \geq b_i$ or $c_{ji} \geq b_i$. Thus the second of the bracketed expressions is nonnegative and $Y^1 - Y^0 \geq 0$. This completes the proof of theorem 4.

Definition

An index i is said to meet the adjoint test with respect to a set of indices P if

$$\sum_{j \in P} (c_{ij} + c_{ji}) \geq b_i.$$

After we find a set of variables for which the optimum value is known, we readjust the constants b_j to the new value b_j',

$$b_j' = b_j - \sum_{i \in A_j'} (c_{ij} + c_{ji}) \quad .$$

If the modified value b_j' of b_j is negative, i.e., the index j meets the adjoint condition with respect to the set A'_j, it indicates that the test of theorem 3 is successful. In that case the corresponding x_j^* is set to one. If, however, the test of theorem 3 fails, the corresponding b_j is replaced by the modified b_j' and we continue our search for optimum values x_j^*, $j \in A$.

Definition

$S_i^{++} \triangleq \{j$: there exists a sequence $i, j_1, j_2, \ldots j_m, j$ such that $j_1 \in S_i^+$, $j_2 \in S_{j_1}^+, \ldots, j_m \in S_{j_{m-1}}^+, j \in S_{j_m}^+\}$.

Example S_1^{++}, S_3^{++}

Let

$S_1^+ = \{3, 4, 5\}$, $S_2^+ = \{6\}$, $S_3^+ = \{2, 7\}$, $S_4^+ = \{9\}$

$S_5^+ = S_6^+ = S_7^+ = S_9^+ = \phi$. $S_1^{++} = \{2, 3, 4, 5, 6, 7, 9\}$

$S_3^{++} = \{2, 6, 7\}$.

Definition

Let $P \subset A$ be a set of indices. A subset $M_j \subset P$ is said to be a mutually covering set (MCS) with respect to P

(i) if $M_j = s_j \cup \{j\}$, $s_j \subset S_j^+$;

(ii) if $i \in s_j$ and $k \in P$, then $i \in S_k^+$ iff $k = j$;

(iii) $j \in S_k^+$ iff $k \in s_j$.

Definition

Let $P \subset A$ be a set of indices; an index j is said to be "covered" in P if there exists an $i \in P$ such that $j \in S_i^+$. If, in addition, there exists a $k \in P$ such that $k \neq j$ and $i \in S_k^+$, then j is "distinctly covered" by i. Finally, if there exists a set $K \subset P$ such that $|K| > 1$ and for each $k \in K$ we have $j \in S_k^+$, then the index j is "multiply covered" by indices of K.

We first prove a few lemmas concerning MCS which will help us show additional sufficiency tests for $x_i^* = 1$.

Lemma 1a

Let $i \in S_i^{++}$. If S_i^{++} does not contain a MCS with cardinality greater than one, then each element $j \in S_i^{++}$ is distinctly covered by some element $k \in S_i^{++}$.

Proof

Let $j \in S_i^{++}$ and assume that j is not distinctly covered. Let $T \subset S_i^{++}$ be the set of indices which cover j. If $|T| > 1$, then j is multiply covered. Now since j is not distinctly covered we must have for each $t \in T$ that $t \in S_j^+$. Now for at least one $t \in T$ there must be a $k \in S_i^{++}$, $k \neq j$ such that $t \in S_k^+$; otherwise the set $M_j = \{i\} \cup T$ is a MCS in S_i^{++} with cardinality greater than one, which is a contradiction. But note that if for some $t \in T$ we have a $k \neq j$ and $k \in S_i^{++}$ such that $t \in S_k^+$, then by definition t distinctly covers j.

Q.E.D.

Lemma 1b

Let M_i and M_j be two MCS with respect to P. If $i \in s_j$ then,

$$M_i = \{i\} \quad \text{if} \quad |M_j| > 2 \quad \text{and}$$

$$M_i = M_j \quad \text{if} \quad |M_j| = 2$$

Proof

In addition to i, let k, $k \neq i$ be an element of M_i. k has to be equal to j, otherwise $i \in s_j$ and $i \in S_k^+$ which contradicts the fact that M_j is a MCS.

In the case that $j = k$, k cannot be in M_i if $|M_j| > 2$. For, there is an $\ell \in M_j$, $\ell \neq i \neq j$, such that $j \in S_\ell^+$ which contradicts that M_i is a MCS.

Q.E.D.

Lemma 2

Let M_j and M_i be two MCS with respect to P, $j \neq i$, then $s_i \cap s_j = \phi$.

Proof

Case 1 $|M_i| = |M_j| = 1$ obvious.

Case 2 $|M_j| > 1$ and $i \in s_j$,

The result follows from lemma 1b.

Case 3 $|M_j| > 1$, $i \notin s_j$, $j \notin s_i$.

Suppose that $s_i \cap s_j \neq \phi$. Then $\ell \in s_i \cap s_j$ $\ell \neq j \neq i$. It follows that $\ell \in S_j^+$ and $\ell \in S_i^+$ which contradicts condition (ii) of the definition of a MCS.

Q.E.D.

Lemma 3a

$$S_i^{++} = \left(\bigcup_{j \in S_i^+} S_j^{++} \right) \cup S_i^+$$

Proof

If $j \in S_i^{++}$ then by definition there exists a sequence $i, j_1, j_2, \ldots, j_t, j$ such that $j_1 \in S_i^+$, $j_2 \in S_{j_1}^+, \ldots, j \in S_{j_t}^+$. There are two cases to consider.

Case 1 $t = 0$; in this case $j \in S_i^+$.

Case 2 $t \neq 0$; in this case $j \in S_{j_1}^{++}$, $j_1 \in S_i^+$.

Hence we have $S_i^{++} \subset \left(\bigcup_{j \in S_i^+} S_j^{++} \right) \cup S_i^+$.

If $j \in \left(\bigcup_{t \in S_i^+} S_t^{++} \right) \cup S_i^+$ then either $j \in \bigcup_{t \in S_i^+} S_t^{++}$ or $j \in S_i^+$. Now if $j \in S_i^+$, then obviously $j \in S_i^{++}$. If $j \in \bigcup_{t \in S_i^+} S_t^{++}$ then $j \in S_t^{++}$, $t \in S_i^+$.

Therefore there exists a sequence $t, j_1, j_2 \ldots j_m, j$ where $t \in S_i^+$, $j_1 \in S_t^+$, $j_2 \in S_{j_1}^+, \ldots, j \in S_{jm}^+$ and $j \in S_i^{++}$. Therefore $(\bigcup_{j \in S_i^+} S_j^{++}) \cup S_i^+ \subset S_i^{++}$.

Q.E.D.

Lemma 3b

$$S_i^{++} = (\bigcup_{j \in S_i^{++}} S_j^{++}) \cup S_i^+$$

Proof

Let $P = (\bigcup_{j \in S_i^{++}} S_j^{++}) \cup S_i^+$ by previous lemma 3a we have $S_i^{++} = (\bigcup_{j \in S_i^+} S_j^{++}) \cup S_i^+$ and since $S_j^+ \subset S_j^{++}$, $S_i^{++} \subset P$. The proof of the lemma is complete if we show that $S_i^{++} \supset P$. Let $k \in P$ then there are two cases.

Case 1 $k \in S_i^+$, then obviously $k \in S_i^{++}$.

Case 2 $k \notin S_i^+$ and $k \in S_j^{++}$ for some $j \in S_i^{++}$;

in this case there is a sequence $i, j_1, j_2, \ldots j_m, j, k_1, k_2, \ldots k_n, k$ such that $j_1 \in S_1^+, \ldots j \in S_{j_m}^+$, $k_1 \in S_j^+, \ldots, k \in S_{k_n}^+$. Therefore we see that $k \in S_i^{++}$ and also $P \subset S_i^{++}$.

Q.E.D.

Lemma 4

If $i \in S_i^{++}$ then there exists a $L \subset S_i^{++}$ such that

$$S_i^{++} = \bigcup_{j \in L} S_j^+$$

Proof

We can prove the existence of such a set L by showing that S_i^{++} is one such set. Let $P = \bigcup_{j \in S_i^{++}} S_j^{+}$. By definition $S_j^{+} \subset S_j^{++}$.

$$P = \bigcup_{j \in S_i^{++}} S_j^{+} \subset \left(\bigcup_{j \in S_i^{++}} S_j^{++}\right) \subset \left\{\left(\bigcup_{j \in S_i^{++}} S_j^{++}\right) \cup S_i^{+}\right\} = S_i^{++}$$

by lemma 3b. The proof is completed if we show that $S_i^{++} \subset P$. Let $k \in S_i^{++}$, i.e., there exists a sequence i, $k_1, k_2 \ldots, k_n$, k such that $k_1 \in S_i^{+}$, $k_2 \in S_{k_1}^{+}, \ldots, k \in S_{k_n}^{+}$. There are two cases to consider.

Case 1 $n = 0$; in this case $k \in S_i^{+}$ and since $i \in S_i^{++}$, $k \in P$.

Case 2 $n \neq 0$; in this case $k \in S_{k_n}^{+}$ and $k_n \in S_i^{++}$ and $k \in P$.

Hence $S_i^{++} \subset P$ and the proof is complete.

Lemma 5

Let $P = S_i^{++} \cup \{i\}$ or $P = S_i^{++}$. A MCS, M_j ($|M_j| > 1$), exists with respect to P only if $i \in S_i^{++}$. If since M_j exists in P, then $i \in M_j$.

Proof

First let us show that if $i \notin S_i^{++}$ then no M_j can exist with respect to P such that $|M_j| > 1$. We first note that j cannot be equal to i. Otherwise, $|M_i| > 1$ implies that there exists a $k \in s_i$ such that $i \in S_k^{+}$ and hence $i \in S_i^{++}$.

So $M_j \subset P$ and $j \neq i$. Notice that $i \notin M_j$, otherwise $i \in s_j$, which implies that $i \in S_j^{+}$, and since $j \in S_i^{++}$, it would imply that $i \in S_i^{++}$.

Let $k \neq j \neq i$ be an element of M_j. Since $k \in S_i^{++}$ there exists a sequence $i, j_1, j_2, \ldots j_m, k$ such that $j_1 \in S_i^+, \ldots j_m \in S_{j_{m-1}}^+, k \in S_{j_m}^+$.

There are two things to be noted. First, $m \neq 0$. Otherwise, $k \in S_i^+$ which cannot be the case because by definition of a MCS if $k \in M_j$, $k \neq j$ then $k \in S_i^+$ if and only if $i = j$. Secondly, $j_m = j$, because again $k \in S_{j_m}^+$ and $k \notin M_j$, $k \neq j$, imply by definition of a MCS that $j_m = j$.

If $j_m = j$ then by definition of M_j, $j_{m-1} \in s_j$, and consequently $j_{n-2} = j$. We continue until we get to j_1. At this point, depending on whether m is odd or even $j_1 = j$ or $j_1 \in s_j$. If $j_1 = j$ then we have $j \in S_i^+$ which implies that $i \in M_j$, which we have already shown not to be the case. If $j_1 \in S_j^+$ and since $j_1 \in S_i^+$, according to the definition of a MCS i has to be equal to j, which we have shown not to be the case. Therefore we have shown that if $i \notin S_i^{++}$ there is no $M_j \subset P$ such that $|M_j| > 1$.

Now let $i \in S_i^{++}$, i.e., $P = S_i^{++}$. Let $M_j \subset P$ such that $|M_j| > 1$. If $j = i$, $i \in M_j$ by definition of a MCS. If $j \neq i$ there is a $k \in M_j$, $k \neq j$; if $k = i$ the second part of the lemma is proved. If $k \neq i$ there exists a sequence $i, j_1, j_2, \ldots, j_m, k$ such that $j_1 \in S_i^+$, $j_2 \in S_{j_1}^+, \ldots, k \in S_{jm}^+$.

If $m = 0$, by the same arguments as previously used, $i = j$. If $m \neq 0$, $j_m = j$ as previously shown. Again by the above arguments either $j_1 = j$ or $j_1 \in s_j$. For the former case $i \in M_j$; for the latter $i = j$. This completes the proof of lemma 5.

Corollary

A set M_j such that $|M_j| > 1$, is a MCS with respect to $P = S_i^{++} \cup \{i\}$ if and only if it is a MCS with respect to S_i^{++}.

Proof

Let M_j be a MCS with respect to $P = S_i^{++} \cup \{i\}$ where $|M_j| > 1$. We claim that M_j is also a MCS with respect to S_i^{++}. We have by lemma 5 that if M_j exists with respect to P that $i \in S_i^{++}$, in this case $S_i^{++} = P$ hence it follows that M_j is MCS with respect to S_i^{++}.

If M_j in a MCS with respect to S_i^{++} then since $S_i^{++} \subset P$ it follows directly that M_j is a MCS with respect to P.

Q.E.D.

In defining MCS with respect to a set S_i^{++} it is possible for sets M_j, M_k of cardinality two to be identical. This can be seen from the following example.

$$S_1^{+} = \{2,3,4\}$$
$$S_2^{+} = \{1,5\}$$
$$S_3^{+} = \phi$$
$$S_4^{+} = \phi$$
$$S_5^{+} = \phi$$

The set $S_1^{++} = \{1,2,3,4,5\}$ and M_1 and M_2 with respect to S_1^{++} are both $\{1,2\}$. Mutually covering sets M_j with respect to a set S_i^{++} are shown to be unique under certain circumstances by lemma 6.

Lemma 6

Let M_j be a MCS with respect to S_i^{++} such that $|M_j| > 1$.

(i) If $|M_j| > 2$ then M_j is the only MCS of cardinality greater than one in S_i^{++}.

(ii) If $|M_j| = 2$ then there are exactly two MCS of cardinality greater than one namely M_i and M_j; furthermore $M_i = M_j$.

Proof

Let M_j and M_k $j \neq k \neq i$ be MCS in S_i^{++} both of cardinality greater than one. According to lemma 5, $i \in M_j$ and $i \in M_k$, i.e., $i \in s_j \cap s_k$. But this is impossible by lemma 2.

If $k = i$, then the lemma follows from the fact that $i \in M_j$ and lemma 1b.

Q.E.D.

We note that if $i \in S_i^{++}$ it is not necessary for a MCS of cardinality greater than one to exist. This is shown by the following example.

Example

$$S_1^{+} = \{2,3,4\} \qquad S_1^{++} = \{1,2,3,4\}$$
$$S_2^{+} = \{1\}$$
$$S_3^{+} = \{2\}$$
$$S_4^{+} = \emptyset$$

No MCS exists in S_1^{++} with cardinality greater than one.

The following lemma is useful in making certain that there exists no MCS M_j, $|M_j| > 1$ in S_i^{++}.

Lemma 7

If for each $j \in S_i^+$, $i \notin S_j^+$, then there can be no $M_j \subset S_i^{++}$ such that $|M_j| > 1$.

Proof

Obviously M_i cannot be in S_i^{++}, $|M_i| > 1$, if for each $j \in S_i^+$, $i \notin S_j^+$. Suppose for $j \neq i$ a MCS M_j exists in S_i^{++} where $|M_j| > 1$. By lemma 5 $i \in M_j$. Now by (ii) and (iii) of definition of MCS we have $j \in S_i^+$ and $i \in S_j^+$, but by hypothesis if $j \in S_i^+$, $i \notin S_j^+$, hence a contradiction. Therefore no MCS M_j exists such that $|M_j| > 1$.

Q.E.D.

Theorem 5

If $i \in S_i^{++}$ and no $M_j \subset S_i^{++}$ exists such that $|M_j| > 1$, then $x_i^* = 1$ for all $j \in S_i^{++}$.

Proof

This follows from lemma 1a, lemma 4, and theorem 4.

Q.E.D.

The next set of Theorems deal with those cases involving S_i^{++} both when $i \in S_i^{++}$ and when $i \notin S_i^{++}$.

Theorem 6

Suppose there exists a MCS, M_ℓ, with respect to S_i^{++} such that $|M_\ell| > 1$. Then if there exists an index

$t \in M_\ell$ such that

$$\sum_{k \in S_i^{++} - S_t^+} (c_{tk} + c_{kt}) \geq b_t,$$

then $x_k^* = 1$ for all $k \in S_i^{++}$.

Proof

We have $i \in S_i^{++}$ and suppose that there exists a MCS, M_ℓ, in S_i^{++} such that $|M_\ell| > 1$. Let $t \in M_\ell$ such that the condition of the theorem holds. Let $Y^1 = y(\underline{x}')$ where

$$x_k' = 1, \qquad k \in S_i^{++};$$

$$x_k' \text{ is arbitrary} \qquad k \notin S_i^{++}.$$

Let $Y^0 = Y(\underline{x}'')$ where

$$x_k'' = 0, \qquad k \in I \subset S_i^{++}$$

$$x_k'' = 1, \qquad k \in S_i^{++} - I;$$

$$x_k'' = x_k' \qquad \text{otherwise}.$$

The theorem will be proved if we can show that $Y^1 - Y^0 \geq 0$ for all $I \subset S_i^{++}$. Consider $Y^1 - Y^0 =$

$$= \sum_{\substack{j \notin S_i^{++} \\ k \in I}} (c_{jk} + c_{kj}) x_j + \left[\sum_{\substack{k \in I \\ j \in S_i^{++}}} (c_{kj} + c_{jk}) - \sum_{k \in I} b_k \right]$$

The terms outside the brackets are nonnegative. We have two cases to consider.

Case 1, $t \notin I$.

We can then write the terms within the brackets as follows:

$$(a)\quad \sum_{\substack{k\in I\cap M_\ell \\ j\in S_i^{++}}} (c_{kj}+c_{jk}) + \sum_{\substack{k\in I-M_\ell \\ j\in S_i^{++}-M_\ell\cap I}} (c_{kj}+c_{jk}) - \sum_{k\in I\cap M_\ell} b_k - \sum_{k\in I-M_\ell} b_k$$

Consider any $k \in I\cap M_\ell$, if $k \neq \ell$ then this index k is "covered" by ℓ, i.e., $c_{k\ell} \geq b_k$ or $c_{\ell k} \geq b_k$. If $k = \ell$, then since $t \in M_\ell$ and $t \notin I$, we can have t as a unique coverer of ℓ, i.e., $c_{t\ell} \geq b_\ell$ or $c_{\ell t} \geq b_\ell$. Now consider any $k \in I - M_\ell$, since $k \in S_i^{++}$ it is clear that there exists at least one $j \in S_i^{++}$ "covering" k, i.e., $c_{jk} \geq b_k$ or $c_{kj} \geq b_k$. We note since there is a single MCS in S_i^{++}, with cardinality greater than one, there is no chance that $\{j,k\}$ form a mutually covering set with respect to S_i^{++}. It therefore follows that expression (a) is nonnegative.

Case 2 $t \in I$.

Now we write the terms within the brackets as

$$(b)\quad \sum_{\substack{k\in I\cap M_\ell \\ j\in S_i^{++} \\ j,k\neq t}} (c_{kj}+c_{jk}) + \sum_{\substack{k\in I-M_\ell \\ j\in S_i^{++}-I\cap M_\ell}} (c_{kj}+c_{jk}) - \sum_{k\in I-\{t\}} b_k + \Big\{ \sum_{j\in S_i^{++}-S_t^+} (c_{tj}+c_{jt}) - b_t \Big\} + \sum_{j\in S_t^+} (c_{tj}+c_{jt})$$

Each $k \in I\cap M_\ell$ $k\neq t$ is covered by index ℓ by the same argument given for case 1. Similarly all indices in $I - M_\ell$ are covered again by the same argument given for case 1. Now t is the only index which is not covered but we see that the terms in the braces are nonnegative by hypothesis. It follows then that expression (b) is nonnegative since t always covers the index ℓ. Q.E.D.

Theorem 7

Let $i \notin S_i^{++}$ and let $\sum_{j \in S_i^{++} - S_i^{+}} (c_{ij} + c_{ji}) \geq b_i$,

then $x_j^* = 1$ for all $j \in S_i^{++} \cup \{i\}$.

Proof

Let $P = S_i^{++} \cup \{i\}$ and let $Y^1 = Y(\underline{x}')$ where

$x_j' = 1$ $\quad j \in P$,

x_j' arbitrary $\quad j \notin P$

Let I be an arbitrary subset of P. Let $Y^0 = Y(\underline{x}'')$ where

$x_j'' = 0$ $\quad j \in I \subset P$,

$x_j'' = 1$ $\quad j \in P - I$,

$x_j'' = x_j'$ $\quad$ otherwise

We claim that $Y^1 - Y^0 \geq 0$ for all $I \subset P$.

$$Y^1 - Y^0 = \sum_{\substack{j \in I \\ k \notin P}} (c_{jk} + c_{kj}) x_k + \left[\sum_{\substack{j \in I \\ k \in P}} (c_{jk} + c_{kj}) - \sum_{j \in I} b_j \right]$$

The terms outside the brackets are nonnegative. It remains to show that the terms within the brackets are nonnegative. We consider two cases; the first case considers the index i to be in the set I. We then write the terms within the brackets as follows.

$$\text{(a)} \quad \sum_{j \in S_i^{+}} (c_{ij} + c_{ji}) + \sum_{j \in S_i^{++} - S_i^{+}} (c_{ij} + c_{ji}) + \sum_{\substack{j \in I - \{i\} \\ k \in S_i^{++}}} (c_{jk} + c_{kj}) - \sum_{j \in I - \{i\}} b_j - b_i.$$

Now by hypothesis we have

$$\sum_{j \in S_i^{++} - S_i^+} (c_{ij} + c_{ji}) \geq b_i;$$

it therefore remains to show that the rest of the expression (a) is nonnegative. We can write

$$\sum_{j \in I - \{i\}} b_j = \sum_{j \in (I-i) \cap S_i^+} b_j + \sum_{j \in I - \{i\} - S_i^+} b_j.$$

Now if $j \in (I-i) \cap S_i^+$ then there exists a $k \in P$, viz i, such that $c_{jk} \geq b_j$ or $c_{kj} \geq b_j$; This j will then be "covered" by a term in the first summation of expression (a).

Suppose $j \in I - \{i\} - S_i^+$, then since by lemma 5 there does not exist a MCS, M_j, with respect to P such that $|M_j| > 1$; we then have j "covered" by a c_{jk} where $k \in S_i^{++}$. It then follows that expression (a) is nonnegative and we have completed the proof for the case $i \in I$.

The next case considers $i \notin I$. In this case

$$Y_i^1 - Y_i^0 = \sum_{\substack{j \in I \\ k \notin P}} (c_{jk} + c_{kj})\, x_k + \left[\sum_{\substack{j \in I \\ k \in P}} (c_{jk} + c_{kj}) - \sum_{j \in I} b_j \right]$$

As before the terms outside of the brackets are nonnegative; the terms within the brackets can be written

$$\text{(b)} \quad \sum_{\substack{j \in I \cap S_i^+ \\ k \in P}} (c_{jk} + c_{kj}) + \sum_{\substack{j \in I - S_i^+ \\ k \in P - I \cap S_i^+}} (c_{jk} + c_{kj}) - \sum_{j \in I \cap S_i^+} b_j - \sum_{j \in I - S_i^+} b_j$$

If $j \in I \cap S_i^+$ then j is "covered" by i. If $j \in I - S_i^+$, then there exists a $k \in P$ such that $c_{kj} \geq b_j$ or $c_{jk} \geq b_j$ and since there is no MCS in P j does <u>not</u> cover k uniquely. It then follows that (b) is nonnegative.

Q.E.D.

Theorem 7 is the last test in the first phase of the algorithm. A flow chart of the algorithm is presented in figure 2.1.

In figure 2.1 we see that the conditions of theorems 1a, 1b, 2, and 3 are tested in the boxes numbered 1, 2, and 3 respectively. In box number 5 we use lemma 7 to facilitate the testing for conditions of theorem 5; this is shown is box 5. The tests of theorems 6 and 7 are represented by boxes 6 and 4 respectively.

2.2 Computational Results Using Phase I

Table 2.1 summarizes the results of the computations for a series of test problems. The coefficients c_{ij} and b_j were randomly selected using a standard random number generator. To avoid trivial solutions all b_j selected were positive. The number of variables for each test problem is designated by N, the number of distinct coefficients c_{ij} in the objective functions is represented by M. The number of variables not found by the first phase of the algorithm is $|A|$. In table 2.1 we list the maximum number not found in any of the five test problems. The times shown represent the running time of the algorithm written in Fortran running on the IBM 360 model 67.

Table 2.1 shows that the average running times ranged from .02 seconds for the 10 variable problems to 9.27 seconds for the 150 variable problems. The growth of the running time with N does not appear prohibitive. Due to the nature of the directed search of the first phase of the algorithm the more significant growth parameter is the number

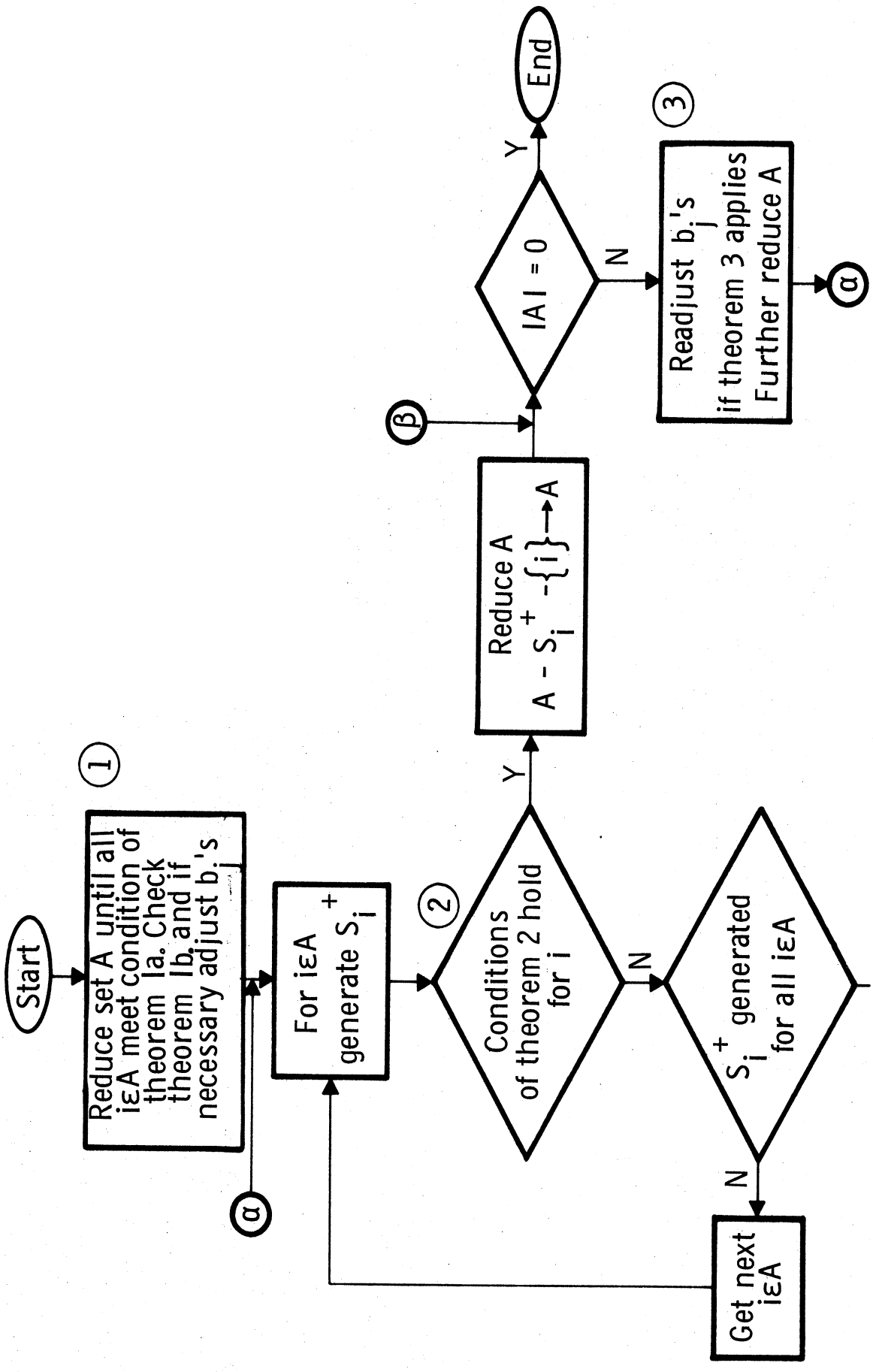

Start
1
Reduce set A until all iεA meet condition of theorem 1a. Check theorem 1b and if necessary adjust b_j's
α
For iεA generate S_i^+
2
Conditions of theorem 2 hold for i
Y
Reduce A
$A - S_i^+ - \{i\} \rightarrow A$
β
|A| = 0
Y
End
N
3
Readjust b_j's if theorem 3 applies Further reduce A
α
N
S_i^+ generated for all iεA
N
Get next iεA

Figure 2.1

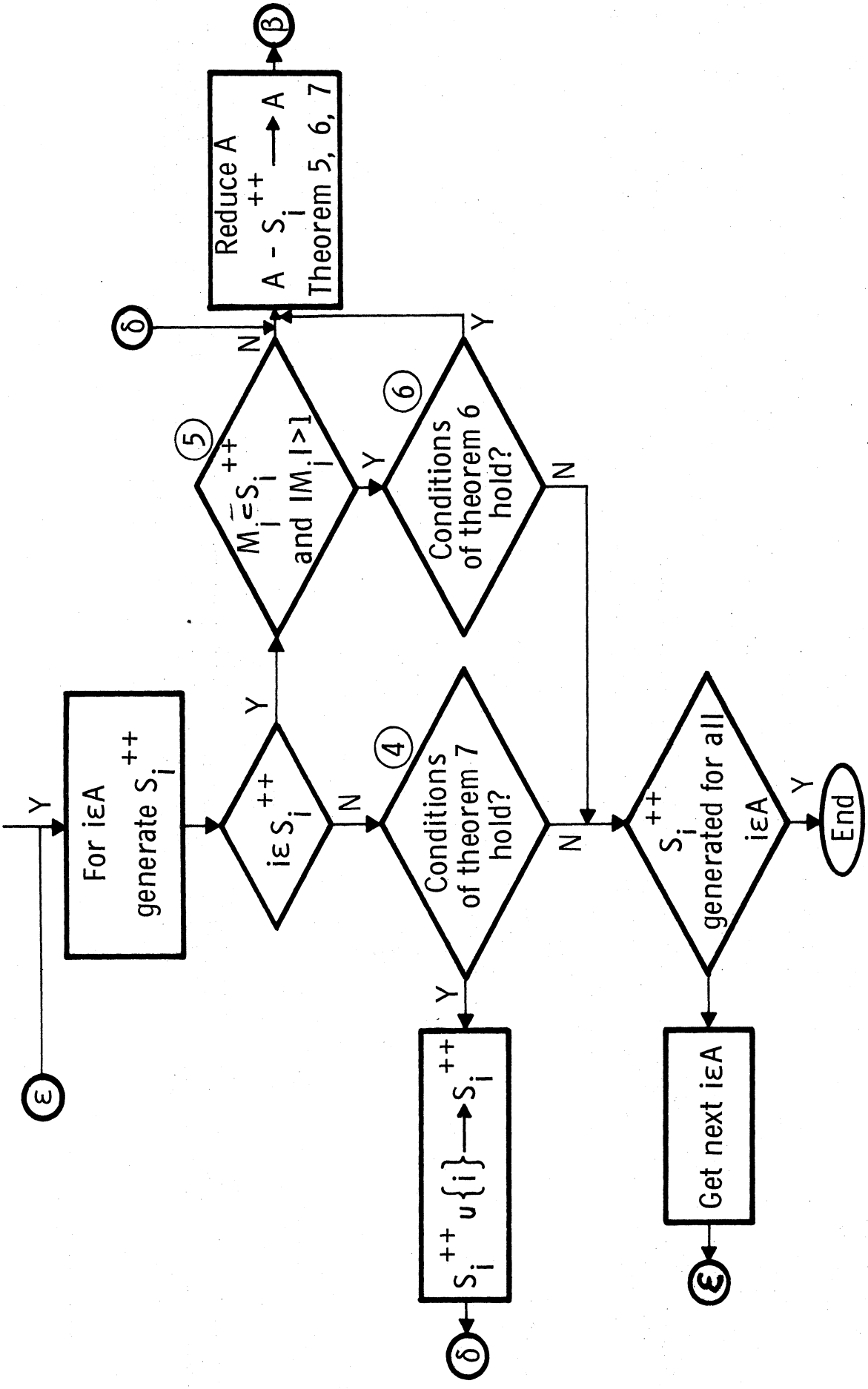


Figure 2.1 (cont.)

N	M	Problems	Average Time (Seconds)	Maximum Time (Seconds)	Max |A|
10	15	5	.02	.05	3
25	50	5	.11	.18	0
50	100	5	.58	.82	2
75	200	5	1.40	1.98	2
100	300	5	2.36	3.84	0
125	400	5	4.43	6.37	2
150	600	5	9.27	12.16	0

Table 2.1*

*After this table was constructed we did generate one example consisting of 150 variables and 500 coefficients where $|A|$ was 17 at the end of phase I.

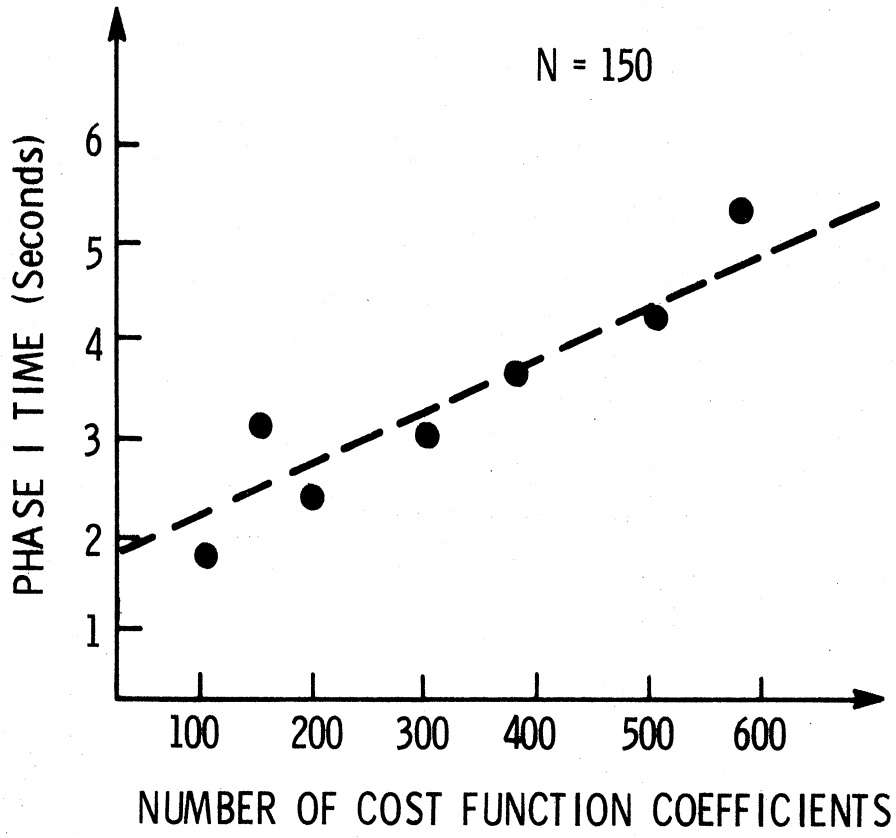

N = 150
PHASE I TIME (Seconds)
1
2
3
4
5
6
100
200
300
400
500
600
NUMBER OF COST FUNCTION COEFFICIENTS

Figure 2.2

of coefficients M in the cost function. For N fixed at 150 figure 2.2 shows the variation of running time with M. The solution times ranged from 1.78 seconds M = 100 to 5.17 seconds M= 600. The dashed curve was fitted to the data points. The time is seen to vary linearly with M. The slope is 5.0 milliseconds per coefficient, which says that for N= 150 about 500 milliseconds in additional phase I solution time is needed for every 100 additional nonzero coefficients in the objective function.

In summary it appears that the algorithm is quite effective in reducing the dimension of a quadratic function as defined by equation (2.1). It seems that the tests are sufficiently powerful to allow use of this algorithm for problems with a large number of (0,1) variables, especially in those cases where the coefficient matrix $[c_{ij}]$ is rather sparse.

2.3 Phase II

The purpose of the first phase of the algorithm is to reduce the dimension of the problem. Once this has been accomplished we begin the second phase of our algorithm. In the second phase we obtain the optimum values for those indices which remain in the set A, therefore the problem with which we deal is

$$\sum_{\substack{i \in A \\ j \in A}} c_{ij} x_i x_j - \sum_{j \in A} b_j .$$

Note that some b_j's have been modified as suggested on page (31),

since many of the variables will be determined in the first phase of the algorithm. Since the dimension of the problem was small after application of phase I of the algorithm, we could use an enumerative procedure for finding the remaining variables. In section 2.5 we discuss an alternative approach when the number of variables is large.

2.3.1 Optimization by Implicit Enumeration

In phase II of our procedure we turn to implicit enumeration to determine the value of the remainder of the variables. We now review a rapid technique for implicit enumeration.

We use a technique for searching the solution space based upon an algorithm of Lawler and Bell [31], developed for purposes of solving discrete variable optimization problem with not a large number of variables. The idea is based on the observation that if the solution vectors can be partially ordered, and the objective function and constraints are monotone nondecreasing functions of the variables, a large number of potential solution vectors can be skipped in a systematic search of the solution space.

2.3.2 Partial Ordering of Binary Vectors

We define a vector partial ordering as follows. Suppose $\underline{\ell} = (\ell_1, \ell_2, \ldots, \ell_n)$ and $\underline{m} = (m_1, m_2, \ldots, m_n)$ are vectors we say

$\underline{\ell} \leq \underline{m}$ if and only if $\ell_i \leq m_i$ for $i = 1, 2, \ldots, n$. For example, if $\underline{\ell} = (10010)$ and $\underline{m} = (10110)$ we have $\underline{\ell} \leq \underline{m}$. We can associate a numerical ordering with a set of binary vectors by associating with each vector $\underline{\ell}$ the corresponding integer

$$N(\underline{\ell}) = \ell_1 2^{n-1} + \ell_2 2^{n-2} + \ldots + \ell_n 2^0.$$

Observe that $\underline{\ell} \leq \underline{m}$ implies $N(\underline{\ell}) \leq N(m)$. However, $N(\underline{\ell}) \leq M(\underline{m})$ does not imply $\underline{\ell} \leq \underline{m}$.

If we list all binary n-vectors in numerical order

$$(0, \ldots, 0, 0)$$

$$(0, \ldots, 0, 1)$$

$$\vdots$$

$$(1, 1, \ldots, 1, 1)$$

We observe that following an arbitrary vector $\underline{\ell}$ there may (or may not) be several vectors $\underline{\ell}'$ with the property $\underline{\ell} \leq \underline{\ell}'$. These are vectors which differ from $\underline{\ell}$ only in that they have 1's in one or more of the right most 0's of $\underline{\ell}$. For example, the vectors following $\underline{\ell} = (0,1,0,0)$ in the numerical ordering are $(0,1,0,1)$, $(0,1,1,0)$, and $(0,1,1,1)$ and each is greater than $\underline{\ell}$ in the vector partial ordering. A vector $\underline{\ell}^*$ is denoted as the first vector following $\underline{\ell}$ in the numerical ordering with the property that $\underline{\ell} \not< \underline{\ell}^*$. For arbitrary $\underline{\ell}$ the vector $\underline{\ell}^*$ is calculated in the following manner:

Consider $\underline{\ell}$ to be a binary number:

(1) Subtract 1 from $\underline{\ell}$

(2) Logically "OR" $\underline{\ell}$ and $\underline{\ell}$-1 to obtain $\underline{\ell}^* - 1$

(3) Add 1 to obtain $\underline{\ell}^*$.

An example

$\underline{\ell} = 10010$

$\underline{\ell}-1 = 10001$

$\underline{\ell}^*-1 = 10011$

$\underline{\ell}^* = 10100$

Note that the vectors in the interval $[\underline{\ell}, \underline{\ell}^*-1]$ are all partially ordered as $\underline{\ell} \leq \underline{\ell}+1, \ldots, \leq \underline{\ell}^*-1$. It is this partial ordering which allows skipping in the enumeration procedure.

2.3.3 Partial Enumeration with Quadratic Function

Consider the objective function

$$Y = \sum_{i,j} c_{ij}x_ix_j - \sum_j b_jx_j.$$

We can write Y as the difference between two nondecreasing functions of $\underline{x} = (x_1, \ldots, x_n)$,

$$Y = g_1(\underline{x}) - g_2(\underline{x})$$

$g_1(\underline{x}) = \sum_{i,j} c_{ij}x_ix_j$, $c_{ij} \geq 0$ $\qquad$ $g_2(\underline{x}) = \sum_j b_jx_j$, $b_j > 0$, since those cases where $b_j \leq 0$ will be eliminated in phase I.

Suppose we examine all 2^n possible solution vectors in numerical order starting with $(0,0,\ldots,0)$ and ending with $(1,1,1,\ldots,1)$. The process can be considerably shortened by using the following observation.

Suppose we test each vector in the numerical ordering keeping track of the best solution so far found. Let $\hat{\underline{\ell}}$ be the solution which has maximized $Y(\underline{x})$. Let $\underline{\ell}$ be the solution currently being considered. We use the following rule to determine if we need test for any solution vector in the interval $[\underline{\ell}, \underline{\ell}^* - 1]$.

If $g_1(\underline{\ell}^* - 1) - g_2(\underline{\ell}) < Y(\hat{\underline{\ell}})$ then skip to $\underline{\ell}^*$. This is because $g_1(\underline{\ell}^* - 1)$ maximizes g_1 in the interval $[\underline{\ell}, \underline{\ell}^* - 1]$, and $\underline{\ell}$ maximizes $-g_2$ in that interval. Therefore it is impossible for some $\underline{\ell}' \in [\underline{\ell}, \underline{\ell}^* - 1]$ to exist such that $g_1(\underline{\ell}') - g_2(\underline{\ell}') \geq Y(\hat{\underline{\ell}})$. Figure 2.3 gives a flow chart of this phase of the algorithm.

2.4 Phase I Computation Efficiency

In this section we analyze the first phase of the algorithm for unconstrained maximization. We attempt to assess the amount of computation involved for a problem consisting of N variables x_j with M distinct nonzero coefficients c_{ij}.

The algorithm begins (see figure 2.1) with tests for conditions of theorems 1a and 1b. For theorem 1a for any variable x_j it will be necessary to examine at most M coefficients for execution of this test. If all variables meet conditions of this test, then the operations

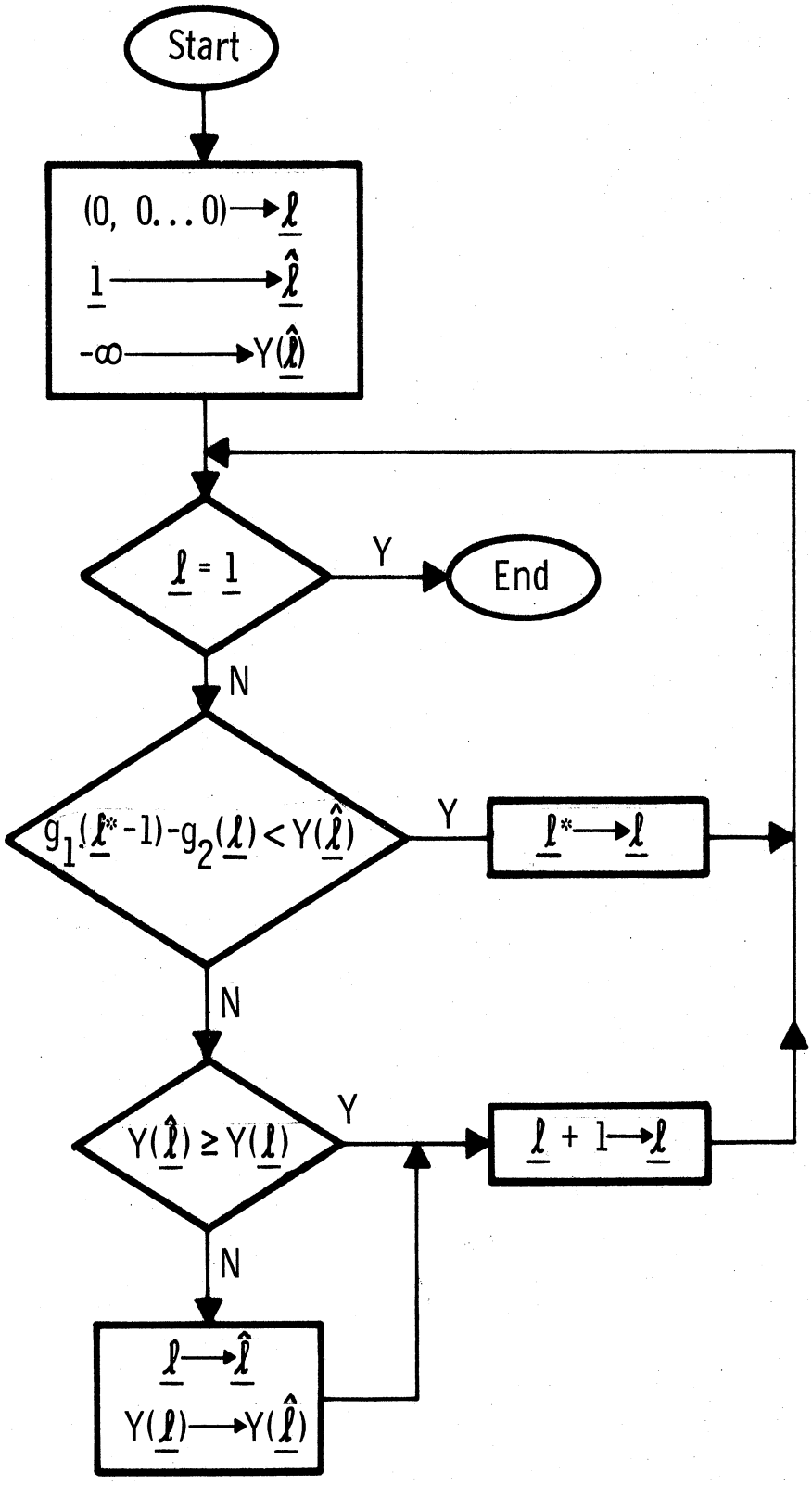


Figure 2.3

are performed once for each variable x_j. In the case that a variable is found for which it is necessary to set to zero by the conditions of the theorem 1a, we are then required to remove all coefficients involving this variable and then recheck the conditions of the theorem for all indices i which comprise the set A. The maximum number of times this must be performed or an upper bound on the passes for this test is N. The total growth then is at most MN^2 for theorem 1a. Table 2.2 summarizes these results.

The check for theorem 1b involves examination of the coefficient b_j for each variable x_j. For any variable this involves a single operation; and it must be performed for at most N variables.

The check for theorem 2 for any variable x_j requires generation of the set S_j^+. Generation of a set S_j^+ requires at most M operations. In box 2 figure 2.1 we check conditions for a given x_j after generation of S_j^+. If conditions are not met we might require this test for at most N variables or we have a maximum of N possible executions of the operations for theorem 2 for a given pass. Whenever we find values for variables via the later tests, theorems 5, 6, 7 we eventually return to the entry labelled α in figure 2.1. We therefore are bounded by N on the number of times we return to perform theorem 2. This gives a total growth rate of MN^2 for this test.

To perform the test for theorem 3 for a given variable it is necessary to perform at most M operations. This test is performed when other tests have found values for variables with indices in the set A. If the test for theorem 3 for some variable x_j is successful, then the test is repeated for all remaining variables in set A. This means that during performance of this test for a pass it might be performed at most K^2 times where K is the number of indices in set A. In table 2.2 we see that a maximum growth for this test is MN^3.

The tests for theorems 5, 6, 7 involve generation of the set S_i^{++}. Essentially a set S_i^{++} for any i where $S_i^+ = \{j_1, j_2, \ldots, j_n\}$ is obtained as follows. We first construct $\Gamma = \bigcup_{t=1}^{n} S_{j_t}^+$. Now for any $\ell \in \Gamma$ such that $\ell \notin S_i^+$ we augment by replacing Γ by $\Gamma \cup S_\ell^+$. We continue until it is no longer possible to augment and this $\Gamma = S_i^{++}$. Construction of an S_j^+ involves at most M operations; this must be done for at most N indices so maximum operations for construction of S_i^{++} is at most MN. The checking for a MCS for a given set S_i^{++} involves examination of at most (N-1) M terms. Therefore the number of operations for the tests for theorems 5, 6 and 7 involve the MN operations for construction of S_i^{++} sets plus at most (N-1) M operations for a check for MCS's and a maximum of M(N-1) operations for theorem 6 and a maximum of N operations for theorem 7. Since this test could be required for N variables

Test	Maximum Number Operations	Upper Bound on Execution Per Pass	Upper Bound on Passes	Rate of Maximum Growth
Theorem 1a	M	N	N	MN^2
Theorem 1b	1	N	1	N
Theorem 2	M	N	N	MN^2
Theorem 3	M	N^2	N-2	MN^3
Theorem 5, 6, 7	NM	N	N-3	MN^3

Phase I Computational Complexity

Table 2. 2

in the set A and repeated at most N-3 times when some test is successful. An upper bound on the growth for these various tests is $N^2M(N-3)$ or operations on the order of MN^3.

2.5 Comparison with Other Methods

There are few other known methods which deal with the unconstrained problem as defined by equation (2.1). Of course the algorithm of Hammer and Rudeanu [19] for general boolean functions could be used for solution of this problem. But this algorithm is highly algebraic and would not be efficient for application to problems containing a large number of variables.

Recently Lawler [30] and then Balinski [4] have discussed a model developed for making optimal selections which could be used as a vehicle for the maximization of (2.1). Their formulation is as follows: Let there be m items i=1, 2, . . . , m; and suppose there are n subsets of the m items $S_1, S_2, \ldots, S_n$. Associate a value d_j with a subset S_j and a cost b_i with an item i. There are no constraints on the collection of subsets $\{S_j, j=1, n\}$ of the m items. It is desired to choose a collection of subsets together with all items which belong to this collection of maximum value. This is done by solving:

Maximize $$\sum_j d_j v_j - \sum_i b_i u_i$$

subject to

$$v_j - u_i \leq 0 \quad \text{whenever } i \in S_j$$

$$u_i, v_j \in \{0, 1\}.$$

$u_i = 1$ if item i is chosen.

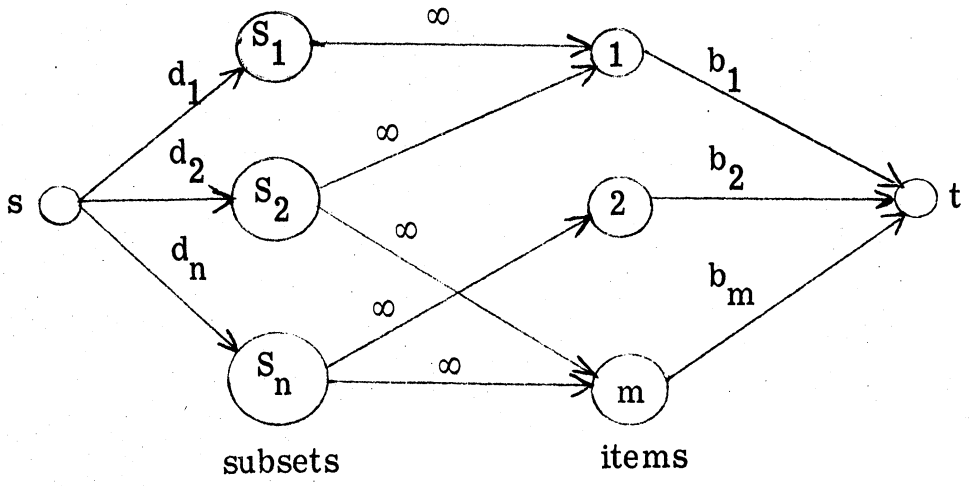
$= 0$ otherwise.

$v_j = 1$ if all items in subset S_j are chosen.

$= 0$ otherwise.

If we define a set S_j for each coefficient $c_{\ell k} > 0$ of (2.1) and assign the value $c_{\ell k}$ to d_j and likewise each coefficient b_i in (2.1) is associated with the cost of item i then the analogy is clear. They then show that this problem can be solved by graphical network techniques.

A directed network is constructed whose nodes consists of a source note s, and a sink node t, and a node for each item i and for each subset of items S_j. The arcs of the network consists of arcs (s, S_j) for each set S_j and (i, t) for each item i. Each arc (s, S_j) is assigned a capacity d_j and arcs (i, t) are assigned a capacity b_j. If $i \in S_j$ then an arc of infinite capacity (S_j, i) is constructed in the network.



A minimum cut of a flow network is defined as a partition of nodes into two sets A and B, where the source node $s \in A$ and the sink $t \in B$. The subset of arcs incident to a node of A and one in B is called a cutset. If these arcs are deleted from the graph all paths between s and t are removed. The value of the cutset is the sum of the capacities on the arcs (i, j) with $i \in A$ and $j \in B$.

It can be shown (see [4]) that the minimum cut of the above flow network corresponds to the selection of the sets from $\{S_j, j=1, \ldots, n\}$ and associated items of maximum value. For such a minimum cut the optimum sets from $\{S_j, j=1, \ldots, m\}$ are identified as those sets S_j which are still connected to s ; the set of nodes still connected to the S_j identifies the items.

We now show how an unconstrained maximization can be obtained using this network flow formulation. We solve the following problem

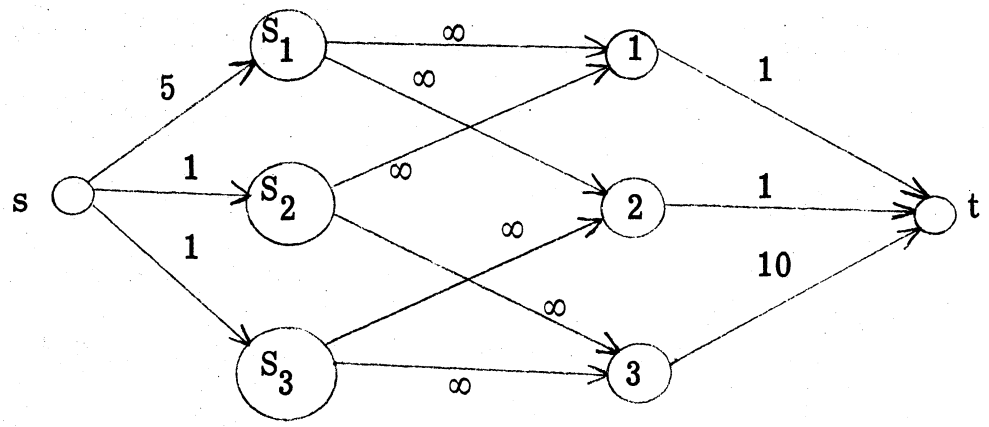
$$\text{Maximize } f = 5x_1x_2 + x_1x_3 + x_2x_3 - x_1 - x_2 - 10x_3$$

If we first solve this problem by our algorithm we get $x_3^* = 0$ by theorem 1a; the problem then is reduced to:

$$\text{Maximize } 5x_1x_2 - x_1 - x_2;$$

which we have $x_1^* = x_2^* = 1$ by theorem 2 and $f^* = 3$.

Formulation of this problem in terms of a flow network gives



$S_1 = \{1, 2\}, \quad S_2 = \{1, 3\}, \quad S_3 = \{2, 3\}$

A minimum cut consists of arcs $(s, S_2), (s, S_3), (1, t)$, and $(2, t)$. Therefore an optimum set of subsets is identified with S_1 since it is still connected to node s and items 1 and 2 which are connected to node S_1. This set has value of 5 - 1 - 1 = 3 which is the same answer we obtained with our algorithm.

The algorithm for finding a minimum cut of network with P nodes has growth on the order of P^5 for arbitrary values of arc capacities (see [30]). In solving (2. 1) with N variables and M coefficients the corresponding network will have N+M+2 nodes. This means the algorithm will have growth rate proportional to $(N+M+2)^5$ for solution of the unconstrained problem.

We found that our algorithm has maximum growth of MN^3 at worst. However, this algorithm does not find all the variables of a

maximizing solution. So in this regard a network solution would be superior. However, in all problems solved the branching algorithm found at least 85% of the variables therefore the problems which remained after application of the algorithm were of trivial dimension.

The network flow solution technique could be used for the second phase of our procedure for unconstrained optimization since it is clearly superior to implicit enumeration.

If the coefficients in (2.1) are limited to integers then the corresponding flow graph problem can be solved with a growth rate proportional to $k(N+M+2)^2$. Clearly this would be a more efficient algorithm for this case.

In conclusion this chapter has presented an algorithm for the optimum solution of a quadratic binary function. The quadratic function arises from problems in operations research. The major part of the algorithm consists in reducing the dimension of the quadratic function which is to be extremized to a size which can be handled by conventional enumerative procedures. The first phase of the algorithm has been found to be very effective in this regard.

Chapter 3

3. Techniques for Constrained Optimization

In this chapter we discuss techniques for solution of constrained optimization problems of the following form,

$$\text{Maximize } f(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j - \sum_{i,j} b_j x_j$$

subject to

$$\text{(A)} \qquad g_i(\underline{x}) \leq \mu_i \quad i=1,2,\ldots,m$$

$$\underline{x} = (x_1, x_2, \ldots x_n)$$

$$x_i \in \{0,1\} \quad i=1,2,\ldots n, \quad c_{ij} \geq 0, \; i,j = 1,2,\ldots n$$

The method used for solution of a given problem defined by (A) will in most cases depend on the nature of the objective function $f(\underline{x})$ and the constraints $g_i(\underline{x})$. This is particularly true when either the objective function or the constraints are non-linear in x_i and/or there are a large number of variables. In this chapter we consider the case where $f(\underline{x})$ is quadratic in x_i and the constraints $g_i(\underline{x})$ are linear.

For a review of techniques involved in solution of (A) when both the objective function and constraints are linear see [3] and [18]; and for a discussion of techniques with no particular restrictions on $f(\underline{x})$ and $g_i(\underline{x})$ see [33].

For the problems with which we were concerned, i.e., m=1 or 2 in problem (A), we found the generalized Lagrange multiplier method of Everett [14] and the family of solutions techniques of Hammer and Rudeanu [19] to be effective when dealing with problems with a moderately large number of variables. With both these methods it is assumed

that one has available techniques for dealing with the free or unconstrained optimization of a function of the same form as $f(\underline{x})$. We use the algorithm developed in chapter 2 for the unconstrained optimization. The generalized Lagrange multiplier method is in a sense more general than the family of solutions technique since with the former there are no restrictions on either $f(\underline{x})$, $g_i(\underline{x})$, or the domain of optimization, while with the latter the constraints $g_i(\underline{x})$ are required to be linear in x_i before the family technique can be applied.

We begin with a discussion of solution of constrained optimization by the method of generalized Lagrange multipliers.

3.1 Generalized Lagrange Multipliers

One effective method for constrained optimization problems is the technique of Lagrange multipliers. A rather recent generalization of this technique to problems containing non-differentiable functions has been made by Everett [14]. We begin by presenting his theorem.

For the statement of Everett's Theorem we first need to state problem B:

(B) maximize $f(\underline{x})$

subject to

$$g_i(\underline{x}) \leq \mu_i \quad i = 1, 2, \ldots m$$

$$\underline{x} = (x_1, x_2, \ldots, x_n) \in X.$$

Everett's Theorem:

For the Lagrangian

$$L(\underline{x}, \underline{\lambda}) = f(x) - \sum_{i=1}^{m} \lambda_i g_i(\underline{x})$$

where $f(\underline{x})$ and $g_i(\underline{x})$ are the same as in problem B and $\underline{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_m)$ is a set of non-negative Lagrange multipliers, if $\underline{x}^* \in X$ is a global maximum for the Lagrangian $L(\underline{x}, \underline{\lambda})$, then $\underline{x}^*$ is the optimum solution of problem B over all $\underline{x} \in \underline{X}$ if μ_i is replaced by $g_i(\underline{x}^*)$ for $i = 1, 2, \ldots, m$. Note that there are no restrictions on the form of the functions $f(\underline{x})$ and $g_i(\underline{x})$ or on the nature of the set X over which the maximization is to be performed.

Proof: Assume that

$$f(\underline{x}^*) - \sum_{i=1}^{m} \lambda_i g_i(\underline{x}^*) \geq f(\underline{x}) - \sum_{i=1}^{m} \lambda_i g_i(\underline{x})$$

for all $\underline{x} \in X$ subject to $\lambda_i \geq 0$ $i = 1, 2, \ldots, m$. We can write

$$f(\underline{x}^*) \geq f(\underline{x}) + \sum_{i=1}^{m} \lambda_i (g_i(\underline{x}^*) - g_i(\underline{x})) .$$

Since for each i we have $\lambda_i \geq 0$ and $g_i(\underline{x}^*) \geq g_i(\underline{x})$; it follows that $f(\underline{x}^*) \geq f(\underline{x})$, which proves the theorem.

If $g_i(\underline{x}^*) = \mu_i$ for each i, then the problem (B) has been solved. However, for arbitrary choices of λ_i's this in general will not be the case. The procedure, then, for the solution of problem (B) is to solve first Everett's problem with arbitrary λ_i's and if all $g_i(\underline{x}^*) \neq \mu_i$, choose

another set of multipliers, optimize the new Lagrangian, and recheck the constraints. The procedure is repeated until for each i we have $g_i(\underline{x}^*)$ sufficiently close to μ_i. The value of the Lagrange multiplier method depends upon how practical it is to optimize the Lagrangian, and on the efficiency of selecting a set of multipliers which give values of constraints which are reasonably close to the given μ_i's.

A second result Everett obtained was that for a given i,if λ_j are fixed for $j \neq i$, $g_i(\underline{x}^*)$ is a monotone nonincreasing function of λ_i. This allows interpolation to be used to select the λ_i which gives the desired value of μ_i. We must add however that there may be values of μ_i such that there does not exist λ_i which will give an $\underline{x}^*$ such that $g_i(\underline{x}^*) = \mu_i$. Everett refers to such a situation as a "gap" and gives some modifications to his basic theorem to determine solutions in such cases.

Figure 3.0 is a schematic of the procedure for solving a constrained optimization problem by the method of generalized Lagrange multipliers. The procedure consists of three steps. First a set of

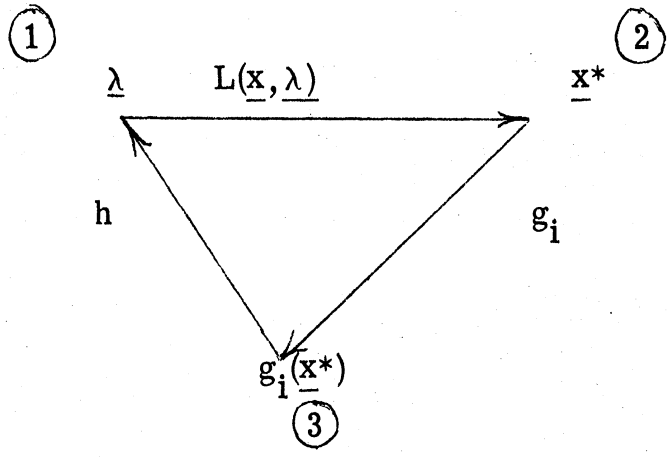


Figure 3.0

nonnegative multipliers is selected. This gives rise to a particular Lagrangian $L(\underline{x}, \underline{\lambda})$. The next step consists of finding $\underline{x}^*$, a vector which maximizes $L(\underline{x}, \underline{\lambda})$. Now each constraint $g_i(\underline{x}^*)$ is evaluated. If each $g_i(\underline{x}^*)$ has the appropriate value, by Everett's theorem, the problem is solved. If not, a new set of multipliers must be selected. This is indicated by the function h which takes the current multipliers $\underline{\lambda}$ and the current values of constraints $g_i(\underline{x}^*)$ and uses this information to select a new set of multipliers. The particular mapping h used for the case of quadratic knapsack problem is defined in section 3.2. This procedure is continued until either a set of constraint values is found which are appropriate or until it is discovered that no $\underline{\lambda}$ exists which will give the values of $g_i(\underline{x}^*)$ within the required range of μ_i. In general it is not possible to say apriori if a set of multipliers exists which will give a solution to a given constrained optimization problem. However, when $f(\underline{x})$ and each $g_i(\underline{x})$ are linear, a necessary and sufficient condition [35] for existence of multipliers which give a solution is that the continuous analog of problem (A), i.e., the problem in which the constraint $x_i \in \{0,1\}$ is replaced by $0 \leq x_i \leq 1$, has an extreme point solution.

3.2 Using Generalized Multipliers for Quadratic Knapsack Problem

In this section we show how multipliers can be used to solve the problem defined in Section 1.2.1 as the quadratic knapsack problem. Recall this problem was defined as:

Maximize $f(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j$

subject to

(A') $$g(\underline{x}) = \sum_i a_i x_i \leq \mu$$

$$x_i \in \{0,1\} \text{ and } a_i > 0 \text{ for all } i, \mu > 0$$

$$c_{ij} \geq 0, \text{ for all } i, j.$$

In this case we deal only with one multiplier since there is only one constraint. The Lagrangian for problem (A') is:

$$L(\underline{x}, \lambda) = \sum_{i,j} c_{ij} x_i x_j - \lambda \sum_j a_j x_j$$

The crux of the generalized multiplier technique is to find the appropriate value of λ if it exists. The appropriate λ is designated as $\bar{\lambda}$ where we define $\bar{\lambda}$ as follows. Given some positive ϵ we have

$$g(\underline{x}^*(\bar{\lambda})) = \bar{\mu}$$

$$0 \leq \mu - \bar{\mu} \leq \epsilon$$

We use a binary search to find the appropriate value of λ. The strategy is to select first an arbitrary λ^0 such that $\underline{x}^*$, the maximizing vector of $L(\underline{x}, \lambda^0)$, gives $g(\underline{x}^*) = 0$. This initial value λ^0 is an initial upper bound for λ; the initial lower bound is zero. The fact that $g(\underline{x}^*(\lambda))$ is a monotone decreasing function of λ makes the search for $\bar{\lambda}$ straightforward. The algorithm used for finding $\bar{\lambda}$ follows:

Step 0. Select $\epsilon > 0$, $\delta > 0$, λ^0 such that $g(\underline{x}^*) = 0$;

$$\lambda_L = 0, \ \lambda_H = 2\lambda^0$$

Step 1. $\lambda = \dfrac{\lambda_H + \lambda_L}{2}$ $\qquad (\lambda_L < \lambda_H)$

Step 2. Find $\underline{x}^*$ which maximizes $L(\underline{x}, \lambda)$.

Step 3 If $g(\underline{x}^*) > \mu$, then go to step 6.

Step 4. If $\lambda_H - \lambda_L \leq \delta$ or $\mu - g(\underline{x}^*) \leq \epsilon$, then set

$\bar{\lambda} = \lambda$, $\mu = g(\underline{x}^*)$, save $\underline{x}^*$, go to step 7.

Step 5. Set $\lambda_H = \lambda$, save $\underline{x}^*$ go to step 1.

Step 6. If $\lambda_H - \lambda_L > \delta$, then set $\lambda_L = \lambda$ go to step 1.

Step 7. If $\underline{x}^*$ has been found, then print $\underline{x}^*$ stop;

Otherwise print "no multiplier solution" and stop.

The above procedure finds $\bar{\lambda}$ if one exists. In those cases where no $\bar{\lambda}$ exists such that we can find a feasible solution, i.e., a "gap" as characterized by Everett, we switch to the family of solutions technique which is discussed in Section 3.4.

The above procedure was used effectively to solve a number of quadratic knapsack problems. We now discuss the more difficult problem when there are two constraints.

3.3 The Two Constraint Case

For some applications, to be considered later, it is necessary to have a pair of constraints as opposed to a single constraint as formulated with the problem in section 3.2. In this case we are concerned with selecting a pair λ_1 and λ_2 of multipliers and the procedure is more complicated. Suppose the constraints for problem (A') are written

$$g_1(\underline{x}) = \sum_i a_i x_i \leq \mu_1$$

$$g_2(\underline{x}) = \sum_i b_i x_i \leq \mu_2$$

The Lagrangian is now

$$L(\underline{x}, \lambda_1, \lambda_2) = \sum_{i,j} c_{ij} x_i x_j - \lambda_1 \sum_i a_i x_i - \lambda_2 \sum_i b_i x_i \ .$$

In this case appropriate values for λ_1 and λ_2 are defined as $\bar{\lambda}_1$ and $\bar{\lambda}_2$, where for some positive ϵ_1 and ϵ_2 we have

$$g_1(\underline{x}^*(\bar{\lambda}_1, \bar{\lambda}_2)) = \bar{\mu}_1$$

$$g_2(\underline{x}^*(\bar{\lambda}_1, \bar{\lambda}_2)) = \bar{\mu}_2$$

and

$$0 \leq \mu_1 - \bar{\mu}_1 \leq \epsilon_1$$

$$0 \leq \mu_2 - \bar{\mu}_2 \leq \epsilon_2$$

We know from Everett's results that if λ_2 is fixed we still have $g_1(\underline{x}^*(\lambda_1))$ as monotone function of λ_1. We therefore select an initial λ^0_2 as some arbitrary large real number. With this value of λ_2 we use the binary search procedure to find $\bar{\lambda}_1$. Now if we find that for this value of $\bar{\lambda}_1$ we have $g_2(\underline{x}^*) = \bar{\mu}_2$ then we stop; otherwise we proceed to select a new value of λ_2 and repeat the binary search procedure for $\bar{\lambda}_1$.

The new value of λ_2 is selected as follows. λ_2 is assumed to be in the interval $(0, \lambda^0_2)$; we initially set $\lambda^2_L = 0$, $\lambda^2_H = 2\lambda^0_2$ where always $\lambda^2_L < \lambda^2_H$, and λ_2 is computed using:

(a) $$\lambda_2 = \frac{\lambda^2_L + \lambda^2_H}{2}$$

As with λ_1, we select subsequent values of λ_2 by using a binary search of the interval $(0, \lambda^0_2)$; if $g_2(\underline{x}^*) > \mu_2$, then the new value of λ^2_L is λ_2 and λ^2_H remains the same. If $g_2(\underline{x}^*) \leq \mu_2$, then the new value of λ^2_H is λ_2 and λ^2_L remains the same. In each case λ_2 is computed using (a).

The effectiveness with which we select the initial arbitrary value for λ_2 will determine the number of iterations necessary for solution of a given problem. We have found that a reasonable initial value for λ_2 is $\sum_{i,j} c_{ij}$, the sum of all coefficients in the cost function. In some of our experimental problems we have been able to decrease the number of iterations by selecting λ^0_2 as some fraction of $\sum_{i,j} c_{ij}$.

Figure 3.1 is a flow chart of the algorithm when there are two constraints. In the flow diagram of Figure 3.1 we have one program exit from a box which says "NO MULTIPLIER SOLUTIONS"; this is necessary since we cannot predict apriori that for all values of the givens $(c_{ij}, a_i, b_i, \mu_1, \mu_2)$ we can find Lagrange multiplier solutions which are feasible. When we encounter this situation we abandon the generalized multiplier approach for constrained optimization.

The next section presents another approach to the problem of constrained optimization.

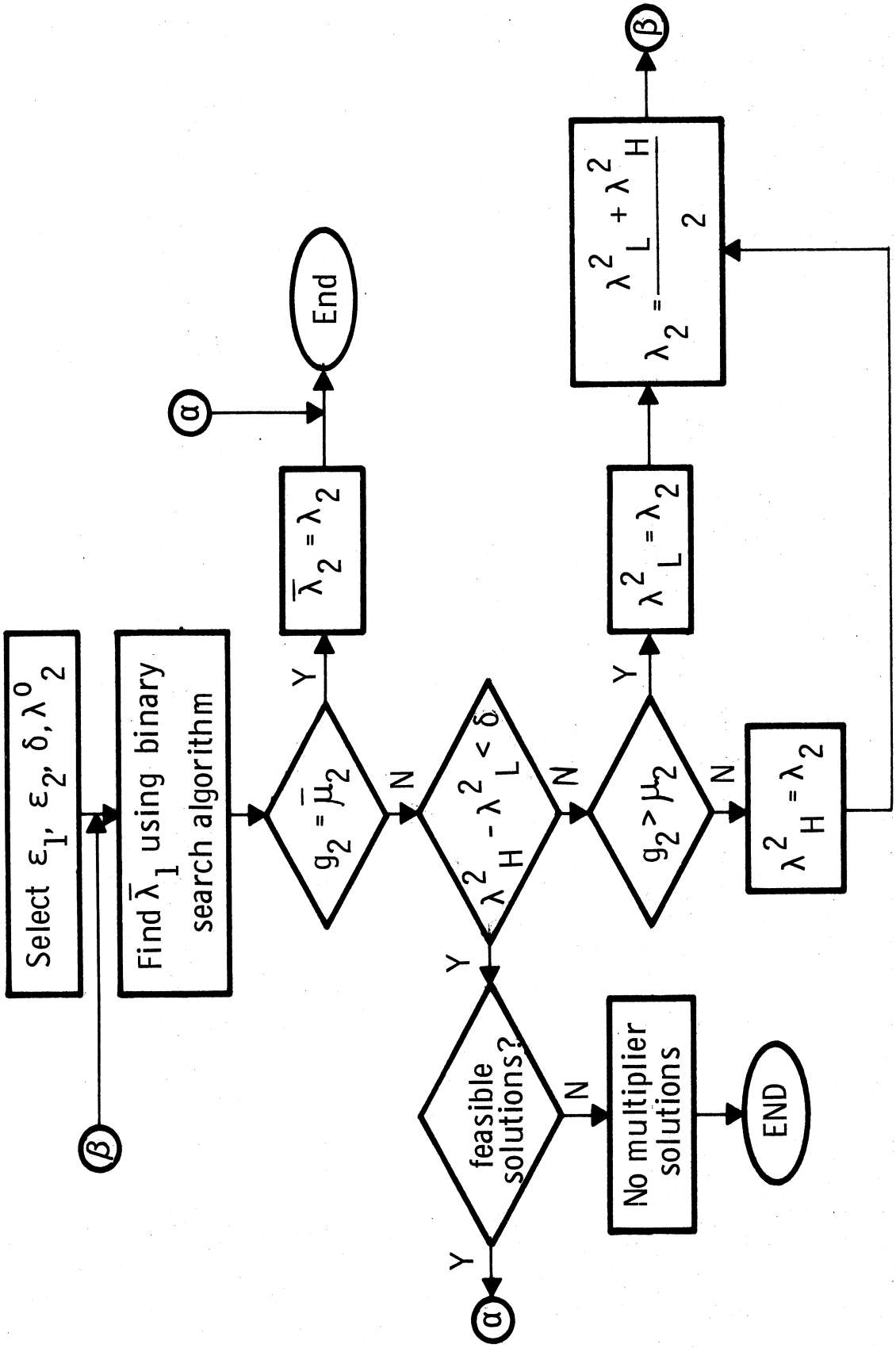

Select ε_1, ε_2, δ, $\lambda^0{}_2$
Find $\bar{\lambda}_1$ using binary search algorithm
$g_2 = \bar{\mu}_2$
Y
$\bar{\lambda}_2 = \lambda_2$
α
End
N
$\lambda^2_H - \lambda^2_L < \delta$
Y
feasible solutions?
Y
α
N
No multiplier solutions
END
N
$g_2 > \mu_2$
Y
$\lambda^2_L = \lambda_2$
$\lambda_2 = \frac{\lambda^2_L + \lambda^2_H}{2}$
β
$\lambda^2_H = \lambda_2$
β

Figure 3.1

3.4 Constrained Optimization Using "Families of Solutions"

Given a constrained optimization problem of the following form,

(1) Maximize $f(\underline{x})$

subject to

(2) $$g(\underline{x}) \leq \mu$$

$$\underline{x} = (x_1, x_2, \ldots, x_n) \quad x_i \in \{0,1\} \quad i = 1,2,\ldots,n \ .$$

A possible procedure for the solution of this problem is to find the set of vectors $S = \{\underline{x}: \ g(\underline{x}) \leq \mu\}$ and from S we select $\underline{x}^*$ such that $f(\underline{x}^*) = \max_{\underline{x} \in S} \{f(\underline{x})\}$. Of course the number of feasible solutions $|S|$ might be quite large; therefore it would be helpful to characterize the solutions of (2) in such a way as to avoid evaluation of $f(\underline{x})$ for every $\underline{x} \in S$. Hammer and Rudeanu [19] have given a procedure for classifying the solutions of a linear inequality into disjoint "families of solutions"; $\{F_1, F_2, \ldots, F_m\}$, such that given any $\underline{x} \in S$, $\underline{x}$ belongs to exactly one family F_i, $i \leq m$. The number of families m is much smaller than the number of feasible solutions $|S|$.

A family F_i is characterized as follows: Let $\hat{\underline{x}} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n)$ be a member of S and let I be a set of indices $I \subseteq \{1,2,\ldots,n\}$, let $\Gamma(\hat{\underline{x}}, I)$ be the set of all vectors $(x_1, x_2, \ldots, x_n) \in \{0,1\}^n = \overbrace{\{0,1\} \times \{0,1\}, \times \ldots \times \{0,1\}}^{n}$ where $x_i = \hat{x}_i$ if $i \in I$, the other variables $x_i (i \notin I)$ are arbitrary. If all vectors $\underline{x} = (x_1, x_2, \ldots, x_n) \in \Gamma(\hat{\underline{x}}, I)$ satisfy (2), then $\Gamma(\hat{\underline{x}}, I)$ is called a family F_j of solutions to (2).

When $\hat{\underline{x}}$ is what is called a basic solution (see appendix I) then an I can be found such that all $\underline{x} \in \Gamma(x, I)$ satisfy (2). The pair $(\hat{\underline{x}}, I)$ is said to generate the family $\Gamma(\hat{x}, I)$; the variables x_i for which $i \in I$ are called fixed variables of the family. When the set I consists of p elements, the family consists of 2^{n-p} solutions, where $p < n$. Hammer and Rudeanu developed an algorithm for finding basic solutions and the sets I associated with each basic solution.

Once the set of disjoint families $\{F_1, F_2, \ldots, F_m\}$ of solutions to (2) is known a procedure for solving (1) and (2) is defined. For each family F_i we find the maximum with our algorithm, described in chapter 2, for unconstrained maximization. The maximum of the problem defined by (1) and (2) is therefore solved as follows:

(1) For each family F_i let $f(\underline{x}^i)$ be the maximum obtained by application of unconstrained optimization algorithm.

(2) $f(\underline{x}^*) = \max_{i=1,2,\ldots,m} \{f(x^i)\}$.

Therefore for a problem with m families we will be required to solve m unconstrained optimization problems. Each unconstrained problem has fewer variables than the original problem since all x_i ($i \in I$) have fixed values. We present in appendix I the algorithm of Hammer and Rudeanu for determining the families of solutions of a linear inequality.

n	Average Time (m.s.)
4	0.5
7	1.0
8	1.1
14	1.5
16	2.6

Table 3.1

A Fortran program running of the IBM 360 model 67 was used to test the speed of the family method. Table 3.1 shows the average time in milliseconds to generate one family for a number of test problems, n is the number of binary variables.

3.5 Example Problem Using Lagrange Multipliers

In this section we give an example of a problem solved using the algorithms thus far described. We want to solve the following problem.

$$\text{Maximize } f(\underline{x}) = 1.40x_1x_3 + .42x_3x_6 + .18x_4x_8 + .42x_5x_8 + .42x_6x_8 + .98x_7x_8 - .7x_1 - .6x_2 - .91x_3 - .09x_4 - .21x_5 - .42x_6 - .49x_7 - x_8$$

subject to

$$g_1(x) = 5x_1 + 9x_2 + 3x_3 + 4x_4 + 7x_5 + x_6 + 9x_7 + x_8 \leq 10$$

$$g_2(\underline{x}) = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \geq 2$$

$$x_i \in \{0,1\} \quad i = 1,8$$

We select for $\epsilon_1 = 2$, $\epsilon_2 = 1$, $\delta = .001$, $\lambda_2^0 = 1.0$. The first step in our program is to find $\bar{\lambda}_1$. In this case $\bar{\lambda}_1 = 0.273$, $g_1(\underline{x}^*) = 9$, $f(\underline{x}^*) = -.79$

$$x_1^* = x_3^* = x_6^* = x_8^* = 1$$
$$x_2^* = x_4^* = x_5^* = x_7^* = 0$$

There were 11 solutions of the Lagrangian for the values of λ_1, which took .17 seconds, using the program for unconstrained optimization developed in chapter 2.

Since $g_2(\underline{x}^*) \neq \bar{\mu}_2$, the second multiplier is adjusted to $\lambda_2 = .5$. This gives the same $\underline{x}^*$ for $\bar{\lambda}_1 = .12$. The binary search technique is used until the proper $\bar{\lambda}_1, \bar{\lambda}_2$ are found, which in this case are:

$$\bar{\lambda}_1 = .089$$
$$\bar{\lambda}_2 = .318$$

which give

$$x_1^* = x_3^* = x_6^* = 1$$
$$x_2^* = x_4^* = x_5^* = x_7^* = x_8^* = 0$$
$$f(x^*) = -.21$$
$$g_1(\underline{x}^*) = 8$$
$$g_2(\underline{x}^*) = 3$$

The values obtained for $\bar{\lambda}_1$ and $\bar{\lambda}_2$ are not unique. The procedure is aimed at finding multipliers which lie in the required intervals. The total number of interations required for solution of this example was 35 with a total time required of .51 seconds. In the sections on applications we will give more computational results obtained with our procedure for constrained optimization problems.

3.6 Comparison of Lagrange Multiplier and Family of Solutions Techniques

The generalized multiplier technique and the family of solution technique were applied to a number of diverse problems. We found advantages and disadvantages with each particular technique.

The main disadvantage of the family of solutions technique is the rapid growth of the number of families. For example if we are finding the number of families for the constraint

$$\sum_{i=1}^{n} a_i x_i \leq \mu$$

where $a_i = 1$ for all i. We see that the number of basic solutions is $\binom{n}{\mu}$. Now in addition to the time required for generation of each of the basic solutions, we must solve each constrained problem associated with a basic solution. The solution time of the unconstrained problem was reduced when using the family technique. This is because, with this method, the number of binary variables in a problem is reduced when we solve for the free maximum. However, for large n we can quickly run up the number of families such that the time to generate the families is prohibitive.

We define an iteration to be the solution of the unconstrained problem for a given set of multipliers. When using the generalized multiplier method, the number of iterations required for solutions was small compared to the number of families generated. Recall that we

must find a free maximum for each family generated. In solving problems with two constraints using the algorithm discussed in 3.3 for a fixed λ_2, the average number of iterations for the search for $\bar{\lambda}_1$ was 14. This was for a set of problems which ranged from 4 to 50 variables.

The main difficulty with multipliers occurs when one encounters "gaps". This phenomena is most likely to occur in problems in which all the given values are integers. The following example illustrates this problem.

3.7 Existence of Gaps Using Multipliers

Suppose we want to solve the following problem with generalized multipliers:

$$\text{Maximize } f(\underline{x}) = 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - 2x_1 - 2x_2 - 2x_3$$

subject to

$$x_1 + x_2 + x_3 \leq 2$$

$$x_1 + x_2 + x_3 \geq 1$$

Feasible solutions will have

$$x_1 + x_2 + x_3 = 2$$

or

$$x_1 + x_2 + x_3 = 1$$

Now the Lagrangian for this problem is

$$L(\underline{x}, \underline{\lambda}) = 2x_1x_2 + 2x_1x_3 + 2x_2x_3 - (2 + \lambda_1 - \lambda_2)x_1 - (2 + \lambda_1 - \lambda_2)x_2 - (2 + \lambda_1 - \lambda_2)x_3 .$$

If we maximize $L(\underline{x}, \underline{\lambda})$ as a function of $\lambda_1 - \lambda_2$, where λ_1 and λ_2 are constrained to be nonnegative, we see that if $\lambda_1 - \lambda_2 > 0$, $\underline{x}^* = (0, 0, 0)$.

On the other hand, when $\lambda_1 - \lambda_2 \leq 0$ $\underline{x}^* = (1,1,1)$. These are the only maximizing solutions for all values of the multipliers. Since both solutions are infeasible we have a gap, and Lagrange multipliers cannot be used for solution of this problem.

The occurrence of gaps in the problems we solved was indeed infrequent. In those cases in which gaps were found the family of solution technique was used. In the chapter on applications more detailed results are presented on the relative frequency of occurrence of gaps.

In conclusion we see that we have rather straightforward ways of converting the constrained problems of the first chapter to an unconstrained form for immediate application of our algorithm for free maximization. In the subsequent chapters we present applications of these techniques to problems of practical interest.

Chapter 4

4. Computer Programming and Graphical Partitioning

In this chapter we consider in detail a particular application of the problem of finding optimal groupings. We consider an example from the area of computer systems storage allocation. This area of application was selected primarily for two reasons. First, as we shall see in chapter six, we had available facilities to collect the necessary statistics for model building; and second the availability of other (chapter five) algorithms or strategies for this particular application allowed assessment of different approaches to the problem.

We first present a model for computer program partitioning and a discussion of the mathematical programming problem derived from this model. A brief discussion is given of the general problem of graphical partitioning of which computer program partitioning is a special case.

We present some sequential partitioning procedures which utilize the solution of the quadratic programming problem, and we close with some computational results which are based on the sequential partitioning procedures.

4.1 Model for Program Graph

In studying the program packing or pagination problem of computer systems attention is focused on the structural and behavioral

properties of programs. In this regard the Markov model [40] is a natural vehicle for description of program structure, i. e., transfers between the various parts of a given program. The frequencies of these transfers or behavioral characteristics can be used in a Markov model for developing a figure of merit for the packing of programs. Perhaps, the most fundamental measure of the effectiveness of a page packing strategy is the resultant number of pages moved between memories per program execution; and since for most systems the decisions relating to traffic between core and secondary devices are highly dependent upon overall system environment, e. g., scheduling algorithms, paging strategies, etc., the effects of packing a program on page traffic are very difficult to ascertain directly. Given these prevailing conditions it is nevertheless true that a program structured to minimize interpage page transfers, reduces the number of systems level decisions relating to that particular program with regards to paging. Therefore it is in this context that the Markov model is used to develop a figure of merit relevant to the average number of interpage transfers per execution of a given program.

4. 1. 1 Program Instruction Units and Data Units

The model used for a program graph is the one proposed in [20]. The salient details of the model are as follows. A program consists of two

collections of units, instruction units $\alpha = \{\alpha_1 \ldots \alpha_m\}$ and data units $\beta = \{\beta_1 \ldots \beta_n\}$. The first collection of units is defined as follows. Each instruction unit $\alpha_i \epsilon \alpha$ is an ordered collection of instruction elements. An instruction element is the smallest functional part of a program, generally a byte or word. Each element, with the exception of exit elements, has associated with it a family of successor elements which are the instruction elements which may be executed immediately after it. An instruction unit is then an ordered collection of elements $e_1, e_2 \ldots e_f$ such that:

1) Each element is an instruction element and appears in exactly one unit;

2) For $1 \leq i < f$, the successor of e_i consists of the single element e_{i+1},

3) For $1 < i \leq f$, e_i is the successor of exactly one element, and that element is e_{i-1}.

4) The total volume of the instruction unit (which equals f bytes if the elements are bytes and size is measured in bytes) is not greater than some fixed maximum.

So we have an instruction unit consisting of a sequence of instructions which may have a single branch point (at the end), and a single entry point (at the beginning), and occupying some number of memory units.

Each data unit $\beta_i \in \beta$ is composed of a set of data elements. Those data elements which can be read or written on by instruction element e_i are said to be referenced by the instruction e_i or referenced by the instruction unit to which e_i belongs. Data units are an unordered collection of elements $b_1 \ldots b_h$ such that:

1. Each element is a data element and appears in exactly one unit;
2. The total volume of the unit (which equals h bytes if elements are bytes and size is measured in bytes) is not greater than some fixed maximum.

Also an entry point and an exit point is specified for the program. If there are multiple entry points and exits one can include dummy units to make the entry and exit unique.

For the problem with which we will be concerned we use the so-called Markov description of a program. Here the matrix $P = \{p_{ij}\}$ describes the transfers of control between instruction units,

p_{ij} = probability that α_j is executed after α_i

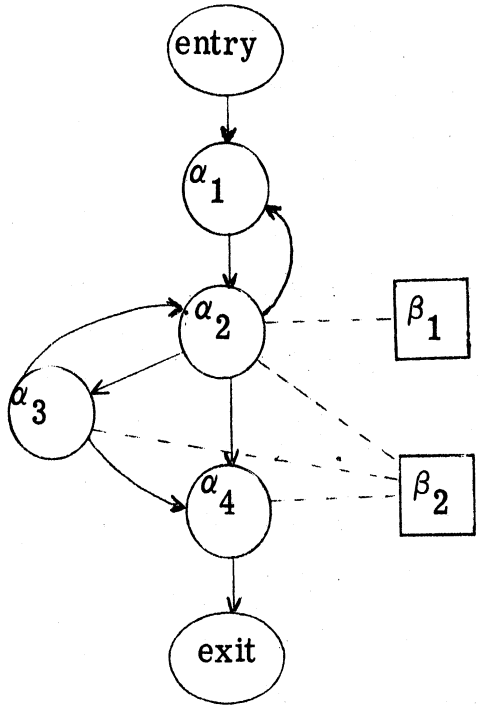
$1 \leq i, j \leq m$ and $\sum_j p_{ij} = 1$ for all i.

For the data units we have the matrix $A = \{z_{iu}\}$,

z_{iu} = probability that α_i references data unit β_u for

$1 \leq i \leq m, \quad 1 \leq u \leq n.$

The following is an example of a program graph along with its associated matrices.



$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1/4 & 0 & 1/4 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad Z = \begin{pmatrix} 0 & 0 \\ .2 & .3 \\ 0 & .5 \\ 0 & .5 \end{pmatrix}$$

The dashed lines indicated that control is not transferred when data units are referenced.

The complete packing problem requires specifying the allocation to pages of both the instruction units and data units. There are various packing strategies one can employ, e.g., mixing of instruction and data units on the same pages, or perhaps having pages consisting of instruction units or data units or both. The point of view we take in this paper is that the pages consists of entirely instruction or entirely data and are not mixed; in this regard we consider only the packing of

the instruction units. The techniques discussed could be applied to pages of data given the appropriate data matrices.

4.2 Figure of Merit for Program Packing

We in this section consider the packing of instruction units only. In order to develop a figure of merit for packing of a program, we use the "frequency of execution" concept developed independently by Kral [27] and at Informatics [20]. Implicit in the objective function, to be developed, will be the number of times each instruction unit is executed. Programs will always have an exit unit; an exit will correspond to an absorbent state of a Markov chain. The remaining units will correspond to the transient states of a Markov chain. It can be shown [23] that if Q represents the transitions between transient states of a Markov chain, then $\{I-Q\}^{-1} = \{n_{ij}\}$ gives the mean number of visits to the transient states of the chain before absorption. That is, n_{ij}, is the mean number of times state j is visited given that the chain started in state i. Therefore if the program is known to start in state one (unit α_1), then we can compute the first row of $(I-Q)^{-1}$ to determine the frequency of execution of each instruction units. It should be noted that Q is derived from the P matrix (defined in the previous section) by deleting from P all rows and columns which correspond to non-transient states. We let γ_i be the number of times unit α_i is executed in one execution of the program. In the case that unit one is the unique initial state then $\gamma_i = n_{1i}$. Let q_{ij} be the probability of a transfer from unit α_i to unit α_j, then $\gamma_i q_{ij}$ is the mean number of transfers from α_i to α_j. Therefore it is the

sum of the quantities $\gamma_i q_{ij}$ over all program units which we desire to minimize by reorganization of the program.

In the next section the mathematical programming formulation is presented.

4. 3 Program Partitioning as a Mathematical Programming Problem

It was noted in the last section that $\gamma_i q_{ij}$ and $\gamma_j q_{ji}$ are measures of transfer activity between instruction units α_i and α_j. The transfer activity of all instruction units is measured by

$$\sum_{i,j} (\gamma_i q_{ij} + \gamma_j q_{ji})$$

Now if a pair of instruction units are on the same page, then transfer between these units are not interpage transfers, these transfers can be tabulated as follows:

$$\sum_{i,j} c_{ij}\, x_{ij},$$

$x_{ij} = 0$, α_i and α_j on same page;

$= 1$, otherwise.

where

$$c_{ij} = \gamma_i q_{ij} + \gamma_j q_{ji} = c_{ji}.$$

It is the quantity $\sum_{i,j} c_{ij}\, x_{ij}$ which is used as a measure of effectiveness for program packing. Since c_{ij} and c_{ji} both appear in the summation we must take one half this quantity, i.e., $\frac{1}{2} \sum_{i,j} c_{ij} x_j$ to get the true number of interpage transfers.

Each instruction unit α_i has a size a_i associated with the number of memory units occupied with the instruction elements. A page of instruction elements is said to be feasible if the instruction units on that page have a total size not exceeding a fixed constant μ which is a size of a physical page. The size constraint can be incorporated as,

$$\sum_{i=1}^{n} (1 - x_{ij})\, a_i \leq \mu \qquad j=1, 2, \ldots, n.$$

This constraint simply adds the sizes of all instruction units which are on the same page; noting of course the reflexive constraint that $x_{ii} = 0$, for all i.

It is necessary to incorporate a transitivity constraint, i.e., if α_i and α_j are on the same page and α_i and α_k are on the same page, then α_j and α_k are on the same page. This is written as

$$x_{ij} + x_{jk} + x_{ik} \neq 2 \quad \text{for all } i, j, k.$$

Finally we have the obvious symmetric constraint $x_{ij} = x_{ji}$.

It therefore follows that a mathematical programming problem representing program pagination is:

(1) Minimize $f(\underline{x}) = \frac{1}{2} \sum_{i,j} c_{ij}\, x_{ij}$

subject to

(I) (2) $\sum_{i=1}^{n} (1 - x_{ij})\, a_i \leq \mu, \qquad j=1, 2, \ldots, n;$

(3) $x_{ij} + x_{jk} + x_{ik} \neq 2, \qquad$ for all i, j, k;

(4) $x_{ij} = x_{ji}, \qquad$ for all i, j;

(5) $x_{ii} = 0, \qquad i=1, 2, \ldots, n;$

(6) $x_{ij} \in \{0, 1\},$

$$\underline{x} = (x_{11}, x_{12}, \ldots, x_{n, n-1}, x_{nn})$$

The problem defined by (1) - (6) is a linear integer programming problem in zero/one variables. The optimum solution of this problem gives the assignment of program units to pages which will result in the minimum mean number of interpage transfers. Consider for a moment the size of this integer linear programming problem. For any program containing n units we will have a problem in $\frac{n^2 - n}{2}$ variables; this means we take advantage of symmetry constraint (4) and of constraint (5) that all $x_{ii} = 0$.

We also have a total of $n + \binom{n}{3}$ constraints from inequalities (2) and (3). So we see that when n is any meaningful size, e.g., greater than 20, the current techniques for solution of linear programming problems in zero/one variables, with the number of constraints indicated, are inefficient. For example, one of the fastest algorithms among existing techniques is the algorithm of Geoffrion [15]. But even here the number of constraints present in the program partitioning problems, as presently formulated, would make application of this algorithm impractical.

We now consider the packing problem in the context of the graphical partitioning problem.

4.4 Graphical Partitioning Problem

It is possible to consider a special case of the program packing problem in the context of the general problem of graphical partitioning which we can state as follows.

Let $G = (V, F)$ be a graph, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices and $F = \{f_1, f_2, \ldots, f_m\}$ is the set of edges. Let $\mathcal{E} = \{F_i : F_j \subset F \; i=1, 2, \ldots n\}$ be a collection of subsets of F, $F_j = \{f_i : f_i$ is incident to $v_j\}$. Let $\mathcal{P}(\mathcal{E})$ be the set of all subsets of $\mathcal{E}$ and let w be a weighing function where $w: \mathcal{P}(\mathcal{E}) \rightarrow R$.

A partition of a graph G is a collection of blocks $C = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_t\}$ such that $\mathcal{E}_i \cap \mathcal{E}_j = \phi$ if $i \neq j$ and $\bigcup_i \mathcal{E}_i = \mathcal{E}$. It is possible to define many different partitioning problems depending of course on the nature of the function w. The graphical partitioning problem is to find a maximal weight partition of the graph G for an arbitrary $w(\mathcal{E})$ where the feasibility of a partition is pre-specified, e.g., a feasible partition might be defined as one in which all blocks $\mathcal{E}_j$ of the partition have $|\mathcal{E}_j| \leq 3$. In the case of optimal packing of program pages, the function w can be defined as

$$(1) \quad w(\mathcal{E}_k) = |\Lambda(\mathcal{E}_k)| - |\Lambda(\mathcal{E} - \mathcal{E}_k) \cap \Lambda(\mathcal{E}_k)|,$$

$$\text{where } \Lambda(\mathcal{E}_k) = \bigcup_{F_j \in \mathcal{E}_k} F$$

The function $w(\mathcal{E}_k)$ counts the number of edges with both ends incident to vertices of $\mathcal{E}_k$. In the context of pagination the vertices of the graph correspond to program units and the edges represent those

transitions between units. We note also that the edge weights are integral valued for this formulation. An additional condition on the formulation of partitions is that of feasibility, i.e., a partition $\mathcal{E}_k$ is said to be feasible if and only if the size of each block $g(\mathcal{E}_k) \leq \mu$. Where the size $g(\mathcal{E}_k)$ is the sum of the sizes of the vertices which make up block $\mathcal{E}_k$.

The weight of a partition C is defined as the sum of the weights $w(\mathcal{E}_k)$ of the individual blocks of the partition. We can minimize the number of interpage transfers by finding the maximum weight feasible partition.

For arbitrary weighing functions $w(\mathcal{E}_k)$ no efficient procedure is known for solution of this graphical partitioning problem. However, for a special class of weighing functions $w(\mathcal{E}_k)$ Lawler [29] has given an efficient procedure, a growth rate proportional to n^5, for the solution to the graphical partitioning problem. Briefly his procedure is to formulate the problem as a problem in dynamic programming; and for the restricted class of weighing functions, the dynamic programming calculations are required only over a very restricted class of subsets of $\mathcal{E}$. One such cost function for which Lawler's results hold is

$$(2) \quad w(\mathcal{E}_k) = |F| - |\Lambda(\mathcal{E} - \mathcal{E}_k) \cap \Lambda(\mathcal{E}_k)|.$$

This weighing function bears a superficial resemblance to the weighing function of program packing, but unfortunately it is not

possible to restrict the class of subsets for (1) as was done with weighing function (2).

4.5 Sequential Selection of Partition Blocks

It appears after one examines the alternative approaches to solving the graphical partitioning problem, especially in the context of program packing, that if graphs of nontrivial order are to be dealt with, one must use some procedure which is suboptimal. In this regard we introduce in this section procedures for partitioning in which the "optimal" blocks of a partition are selected sequentially. In doing so we obtain a resultant partition which will in general be suboptimal. However, for a number of test problems we found this procedure to give optimal results.

4.5.1 Selection of Maximum Weight Partition Blocks

As previously stated, concomitant with the minimization of the inter-block weight is the maximization of the sum of the weight of the individual blocks which comprise the partition. One might be inclined to use as a heuristic strategy a partitioning which selects the blocks with the largest weight. That is, we select from our graph a maximum weight feasible block V_1; a block such that the sum of the weights of the edges with both ends incident to vertices in V_1 is maximum over all feasible blocks in G. This will be the first block of our partition. Now from the graph $G-V_1$ we repeat this procedure for V_2 the second

block of our partition. We continue in this manner until we have a set of blocks $\{V_1, \ldots, V_m\}$ which comprise a complete partition of the graph G. The inter-block weight of this partition will in most cases be close to the inter-block weight of the optimum partition.

The selection of the block V_k with maximum weight can be obtained by solution of the following optimization problem.

$$\max \; f(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j$$

subject to

$$\text{(II)} \qquad \begin{aligned} &\sum_i a_i x_i \leq \mu_1 \\ &x_i \in \{0,1\} \quad i = 1, 2, \ldots, n \\ &x_i = 1 \quad v_i \in V_k, \\ &\quad = 0 \quad \text{otherwise,} \end{aligned}$$

c_{ij} weight of edge (v_i, v_j)

a_j size of vertex v_j

μ_1 maximum allowable block size.

This problem, even though nonlinear, can be solved with the methods described in chapters two and three. There are some obvious advantages to this formulation. First we deal with n variables as opposed to $\frac{n^2 - n}{2}$ and we have only one constraint as opposed to constraints on the order of n^3 as with the linear formulation.

Successive applications of problem (II) were found to give in many cases an optimal partitioning when applied to small test problems for which the optimal results were known. However we could construct examples for which the process of successive selection of the heaviest feasible block could lead to a very poor partitioning.

To assess the computational effectiveness of block selection by solution of (II) a number of tests problems were solved. Table 4.1 summarizes the results.

Column N represents the number of variables and M the number of distinct coefficients c_{ij} in the objective function. The coefficients c_{ij} and a_j were selected using a random number generator from (0, 10). The value of μ_1 was taken as $.4 \sum_{i=1}^{N} a_i$. The column labeled T represents the time taken to select an optimal block form of graph of N vertices and M edges.

N	M	T	D
10	15	.097	.34
20	40	.71	.25
30	60	5.2	.14
40	90	9.6	.12
50	107	13.3	.09

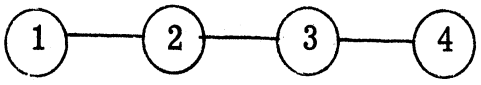
Table 4.1

In the program written to solve the problem we use an upper triangular matrix. That is if there are terms $c_{ij}\, x_i x_j$ and $c_{ji}\, x_i x_j$ in the objective function where $i < j$ we store the value c_{ij} where $c_{ij} = c_{ij} + c_{ji}$. The value of M represents the number of terms in the upper triangular matrix. Since we have $c_{ii} = 0$ for all i, the density of the matrix $[c_{ij}]$ is $\frac{2M}{N^2 - N}$. Column D represents the density for each value of N. For the value of density given the solution time seems to grow as $N^{5.2}$.

4. 5. 2 Selection of Partition Blocks with Minimum Interblock Weight

An alternative procedure for selection of the blocks of a partition which maximize the sum of the weights of the individual blocks is now examined.

In using problem (II) for block selection, poor partitions could result if the blocks which were selected tended to leave the remaining graph highly disconnected. For example consider the following graph, where all vertices are unit size and $\mu = 2$.



If we use the problem (II) as a vehicle for block selection, we could obtain vertices $\{2, 3\}$ as the first block of a partition, since this set is a maximum weight feasible set of vertices. It is clear however that either $\{1, 2\}$ or $\{3, 4\}$ is a superior set for the first block of the partition since in these cases we leave the remaining graph connected.

To anticipate this type of graphical structure we can reformulate the strategy for sequential block selection as follows. Rather than having the weights of the edges with both ends incident to vertices in V_k (the selected block) maximized, we can select V_k such that the weight of the edges between V_k and $V - V_k$ is minimized. This of course is equivalent to maximizing the sum of the weights of the edges with both ends in V_k and with both ends in $V - V_k$. In this case our objective function would be

$$f(\underline{x}) = \sum_{i,j} c_{ij} x_i x_j + \sum_{i,j} c_{ij} (1 - x_i) (1 - x_j) .$$

The second term weighs the edges on the graph after the block of vertices V_k is removed. We can rewrite f(x) as

$$f(\underline{x}) = 2\sum_{i,j} c_{ij} x_i x_j - \sum b_j x_j + K$$

$$b_j = \sum_j (c_{ij} + c_{ji})$$

$$K = \sum_{i,j} c_{ij}$$

Now the quadratic programming problem for sequential block selection becomes:

Maximize $f(\underline{x}) = 2\sum c_{ij} x_i x_j - \sum_j b_j x_j$

Subject to

(III) $g_1(\underline{x}) = \sum_i a_i x_i \leq \mu_1$

$$g_2(\underline{x}) = \sum_i x_i \geq \mu_2$$

$$x_i \in \{0,1\}$$

$$x_i = 1, \quad v_i \in V_k$$

$$= 0, \text{ otherwise, } \mu_2 \geq 1.$$

Notice that we have added the constraint $\sum_i x_i \geq \mu_2$, this constraint is necessary to avoid the trivial solution $x_i = 0$ for all i, which will indeed maximize $f(\underline{x})$ and be feasible without the second constraint.

Problem (III) was found for a number of small graphs to give better results than problem (II) when used as a tool for selection of the optimal blocks of a partition. Note that the constraint K is dropped from the objective function in problem (III) since it plays no role in the maximization of $f(\underline{x})$.

Next we consider some of the problems encountered in using a sequential selection strategy when problem (III) is the vehicle for block selection.

4. 5. 3 The Number of Vertices on Selected Block as a Parameter

For any graph of N vertices, the minimum number of partition blocks is

$$\text{MINPAG} = \left\lceil \frac{\sum_{i=1}^{N} a_i}{\mu_1} \right\rceil^*$$

The maximum number of vertices on a block is

$$\text{MAXVER} = N - \text{MINPAG} + 1$$

$\lceil a \rceil^*$ = smallest integer greater than or equal to a

If $\underline{x}^*$ is an optimal solution to problem (III) then for any feasible two block partitioning we will have $g_2(\underline{x}^*)$ bounded,

$$1 \leq g_2(\underline{x}^*) \leq \text{MAXVER} .$$

In our procedure we select a value, MINVER, as the minimum number of vertices on any block to be removed. We choose MINVER as follows:

$$\text{MINVER} = \text{MAXVER} - \text{CON}, \text{ if MAXVER} \geq \text{CON} + 1$$

$$= 1, \text{ otherwise}$$

The value of CON is preset in the algorithm. Of course by controlling the number of vertices per block we control the number of blocks in a partition.

Recall that in using the Lagrange multipler method of solution we selected the values of the multiplier arbitrarily and used a binary search to obtain intervals for λ_1 and λ_2 such that the constraints were satisfied. Now with problem (III) we know that μ_1 represents the physical limitation on the sum of the sizes of the selected vertices; but μ_2 is only specified to be greater than one. In other words we do not know in advance what the number of vertices should be which will give an optimum two block partition when we solve (III). In fact when we select MINVER as the smallest number of vertices we will remove, Everett's Theorem only tells us that if our block has size $g_1(\underline{x}^*)$ and it contains $g_2(\underline{x}^*)$ = MINVER vertices, then no block of size less than $g_1(\underline{x}^*)$ containing at least $g_2(\underline{x}^*)$ vertices exists which will give a higher $f(\underline{x}^*)$. Therefore before we select a block for our partition we solve for several values of $g_2(\underline{x}^*)$. That is we solve our problem for some

initial set of multipliers such that $g_2(\underline{x}^*)$ is as large as possible, we call this first value $g_2(\underline{x}^*)_{max}$. Then we solve the problem for values of $g_2(\underline{x}^*)$ in the range MINVER $\leq g_2(\underline{x}^*) \leq g_2(\underline{x}^*)_{max}$.

The value selected for λ_2 for the first solution of the Lagrangian is arbitrary. We want to select the initial λ_2 such that the initial number of vertices removed is as large as possible. For any value of λ_2 selected $g_2(\underline{x}^*)_{max}$ is bounded above by MAXVER. We found experimentally that after λ_2 is larger than some constant, λ_{2max}, that the same resulting partition was obtained for all values of $\lambda_2 \geq \lambda_{2max}$. A value we found for λ_{2max} was $\max_j\{b_j\}$ where b_j is the coefficient of the linear term in our objective function $f(\underline{x})$.

After a partition is obtained with this initial value of λ_2 we have associated with this partition $g_2(\underline{x}^*)_{max}$ vertices. Now we look for a partition with $g_2(\underline{x}^*) = g_2(\underline{x}^*)_{max} - 1$ vertices. This is done by using a binary search of the interval (o, λ_{2max}) for a value of λ_2 that gives $g_2(\underline{x}^*)_{max} - 1$ vertices. When the proper λ_2 is found, i.e., it gives the appropriate $g_2(\underline{x}^*)$, we record the value of $f(\underline{x}^*)$. Now the interval (o, λ_2) is used to find the next value of the second multiplier; this is the value which gives $g_2(\underline{x}^*) - 1$ vertices. This procedure is continued until we have $g_2(\underline{x}^*)$ = MINVER. The partition selected is the one with vertices in the range of (MINVER, $g_2(x^*)_{max}$) with maximum $f(\underline{x}^*)$.

So rather than treating block size μ_1 and the number of vertices on a block as fixed known quantities, they are treated as parameters in as much as we look at the value of the objective function for different combinations of block size $g_1(\underline{x}^*)$ and number of vertices $g_2(\underline{x}^*)$. Therefore in this case, where μ_2 is an unknown the algorithm is as follows:

Step 0 Select λ_{2max}, MAXVER, MINVER, $\lambda_{2L} = 0$, $\lambda_2 = \frac{\lambda_{2L} + \lambda_{2max}}{2}$.

Step 1 For this λ_2, find λ_1 such that a block of maximum feasible size is selected. Record $g_2(\underline{x}^*)$. Set $g_2(\underline{x}^*)_{max} = g_2(\underline{x}^*)$. If no feasible $\underline{x}^*$ is found go to step 8.

Step 2 $\mu_2 = g_2(x^*) - 1$.

Step 3 $\lambda_2 = \frac{\lambda_{2L} + \lambda_{2\,max}}{2}$.

Find λ_1 such that we obtain maximum size $g_1(\underline{x}^*)$ feasible block; otherwise if no feasible $\underline{x}^*$ is found go to step 8.

Step 4 If $g_2(\underline{x}^*) = \mu_2$, go to step 7.

Step 5 If $g_2(\underline{x}^*) > \mu_2$, $\lambda_{2max} = \frac{\lambda_{2max}}{2}$ and go to step 3.

Step 6 If $g_2(\underline{x}^*) < \mu_2$, $\lambda_{2L} = \lambda_2$ and go to step 3.

Step 7 If μ_2 = MINVER Stop; otherwise $\mu_2 = g_2(\underline{x}^*) - 1$, $\lambda_{2max} = \lambda_2$, go to step 3.

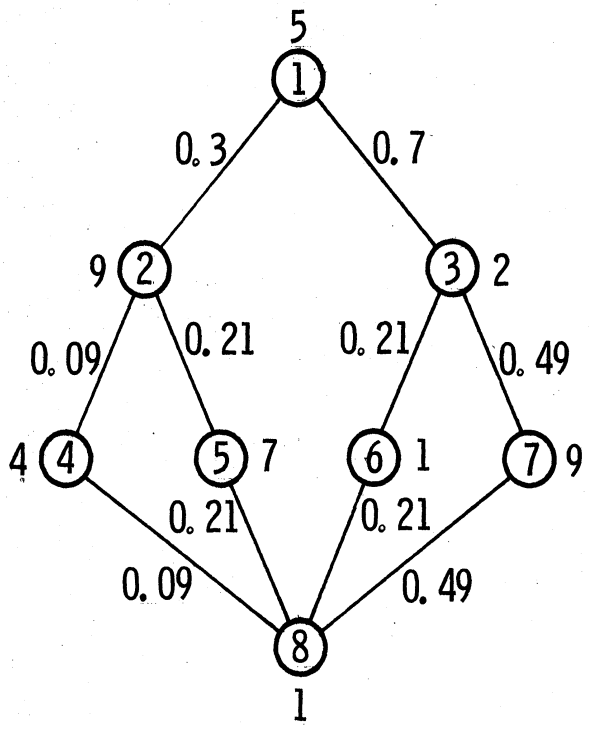
Step 8 Stop multiplier search gap exists, go to family method.

After an optimum set of vertices V has been removed, the procedure is repeated on the graph G - V. For the graph of figure 4. 1, table 4. 2 given the result of the parametric partitioning.

λ_1	λ_2	$g_1(\underline{x}^*)$	$g_2(\underline{x}^*)$	$f(\underline{x}^*)$
0. 2727	1. 0	9	4	-0. 79
0. 120	0. 545	9	4	-0. 79
0. 0898	0. 3175	8	3	-0. 21

Table 4. 2

When λ_2 is set to 1. 0 our binary search procedure selects values for λ_1 until the value λ_1 = 0. 2727 is found. This corresponds to a feasible block of size 9 containing 4 vertices. The value $f(\underline{x}^*)$ represents the weight of the edges between the two blocks. When we selected λ_2 to be . 3175 we get a λ_1 of .0898, which corresponds to a feasible block of size 8 containing 3 vertices. Smaller values for λ_2 give no improvement when MINVER is two as in this case. In this particular case it turns out that the optimum two block partition corresponds to λ_1 =. 0898 and λ_2 = . 3175, which gives a between block edge weight $f(\underline{x}^*)$ of -0. 21. To get the total weight of edges with both ends incident to vertices on the same block, we add a constant K representing the total arc weight of the original graph to the negative number $f(\underline{x}^*)$. The block of vertices removed from figure 4. 1 is $\{1, 3, 6\}$.



Block size = 10

Numbers beside the vertex represent the vertex size

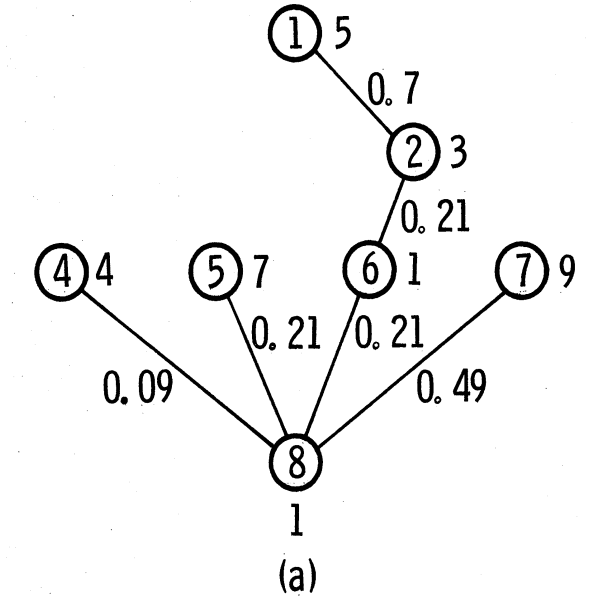
Figure 4.1

4. 6 Algorithm for Graphical Partitioning

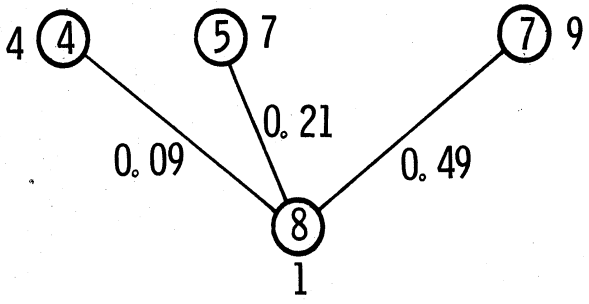
The approach taken is to use the generalized multiplier technique for partitioning the graph whenever possible. If during the process of partitioning a graph a gap in the values for multipliers is discovered we switch to the family of solutions technique. We can state that during the actual running of the algorithm on graphs where all the vertices were not the same size, gaps occurred only when there were very few vertices on the infeasible graph (graph which remained to be partitioned after removal of some vertices). For cases where not all vertices were the same size, we found when it was necessary to use the family method, that the largest number of families generated for a single partitioning was 87, this occurred when the total number of families generated was 115 (see table 4.4 n = 46). Figure 4. 3 is a flow chart of the partitioning algorithm.

4. 6. 1 Example of Partitioning

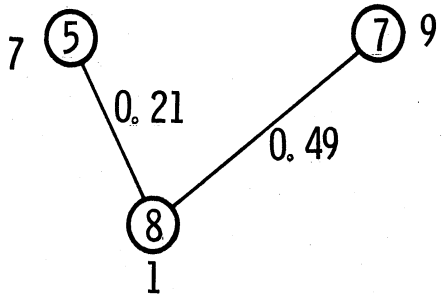
We illustrate the steps in application of this algorithm by finding the optimum partition for the graph of figure 4. 1. First the bounds on μ_1 the maximum block size is set; this is shown in box 1. of Figure 4. 3. In this box we also read CON which determines MINVER. In box two we see all infeasible connections are removed, i. e. , if vertices v_i and v_j have edge c_{ij} between them and $a_i + a_j > \mu_1$, edge c_{ij} is deleted from graph. The edge c_{ij} is called infeasible because it is impossible,



(a)



(b)



(c)

(d)

Figure 4.2

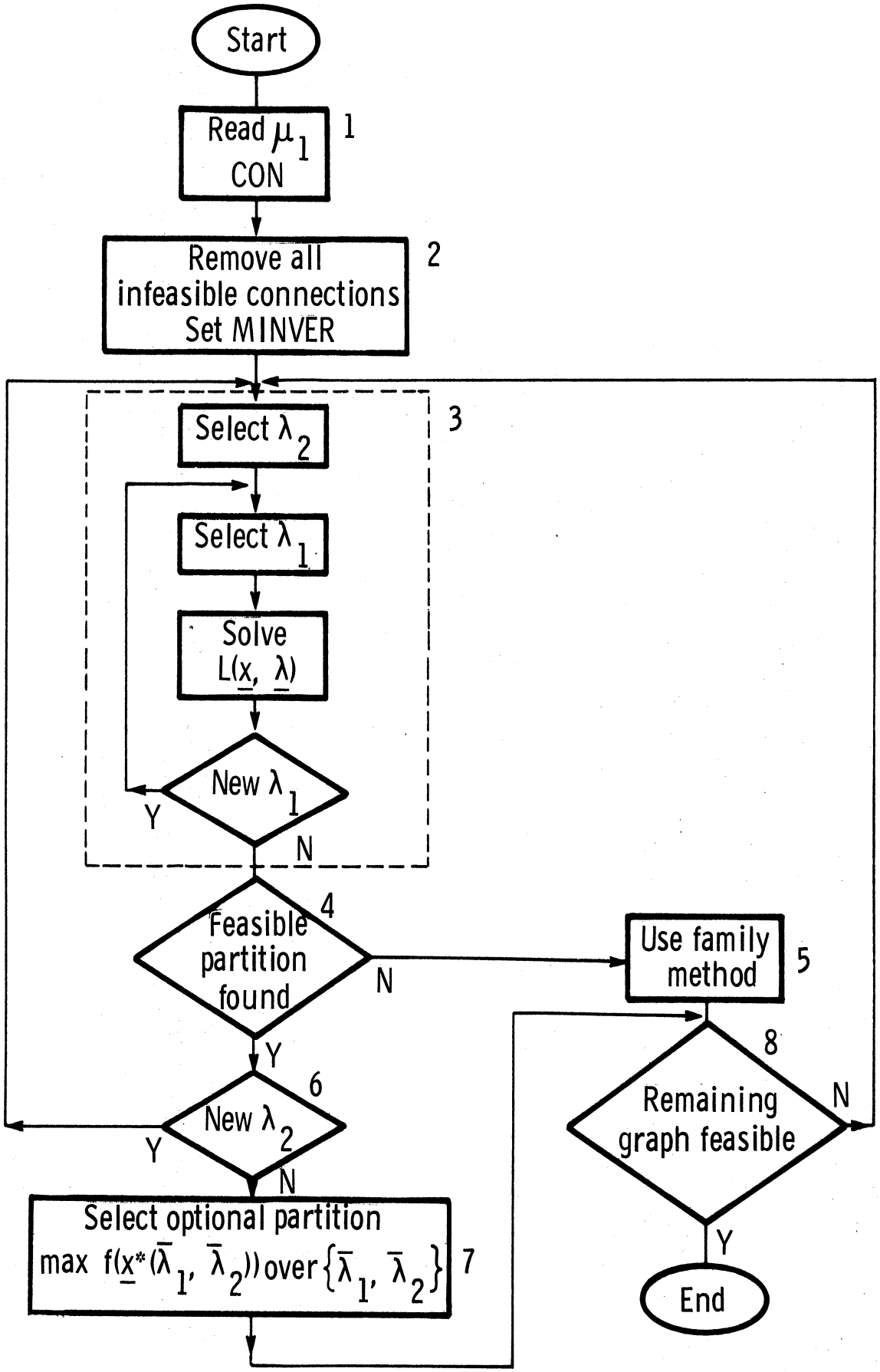

Start
Read μ_1 CON
1
Remove all infeasible connections Set MINVER
2
Select λ_2
3
Select λ_1
Solve $L(\underline{x}, \underline{\lambda})$
New λ_1
Y
N
Feasible partition found
4
N
Use family method
5
Y
New λ_2
6
Y
N
Select optional partition
max $f(\underline{x}^*(\bar{\lambda}_1, \bar{\lambda}_2))$ over $\{\bar{\lambda}_1, \bar{\lambda}_2\}$
7
8
Remaining graph feasible
N
Y
End

Figure 4.3

due to size constraints, for the vertices connected by this edge to be on the same block. The sum of all infeasible edges is a lower bound on the inter-block weight of any partitioning of the graph. In box 2 the lower bound MINVER is calculated from formula MINVER = MAXVER - CON; MAXVER in this case is 5 and CON is 2.

Figure 4.2 (a) represents the graph with which we deal after the checks of box 2 of our algorithm have been applied. Notice that vertex 2 is not on the graph since it is isolated after infeasible edges are removed and no longer plays a role in the optimum partitioning.

The first value of λ_2 is selected. Then the interpolation scheme, i.e., binary search is used to find the appropriate value for λ_1. This sequence is indicated by box 3. After a maximum size $g_1(\underline{x}^*)$ feasible block has been found with the given λ_2 using a binary search of values for λ_1, we consider a new value for λ_2. This is box 6. If $g_2(\underline{x}^*) \neq$ MINVER, the search procedure previously indicated is used to select a new value for λ_2, i.e., first step in box 3 of figure 4.3.

Table 4.3 indicates the optimum values found after application of procedure to graph of figure 4.2(a). Figure 4.2(b) is the subsequent graph after removal of optimum block. Table 4.2 gives the values found after application of procedure to graph of figure 4.3(b). The resulting graph is shown in figure 4.2(c). When we apply the procedure to this graph we note that the multiplier method fails, i.e., a "gap" is encountered.

In this case we use the family method, box 5 in flow chart, to get the optimum block. The resulting graph figure 4. 2(d) is feasible and the partitioning is complete.

λ_1	λ_2	$f(\underline{x}^*)$	Vertices Removed
. 089	. 317	-. 21	1, 3, 6
. 083	. 44	-. 09	4
Gap	Gap	-. 21	7, 8

Table 4. 3

4. 6. 2 Results for Some Partitioning Problems

Table 4. 4 summarizes statistics for a set of partitioning problems solved using the sequential optimization procedure, when problem (III) is used for block selection.

The problem for n = 4, 8, 14 were problems for which the optimal results were known and obtained. The two larger problems were selected by using a random number generator for values of c_{ij} in the interval (0, 10). The values of vertex sizes were randomly selected integers in the range $[1, \mu]$. The table shows that for the largest problem solved a total solution time of 4. 80 seconds. This time is the sum of T1, the total time to generate all families when a gap is encountered and, T2, the total time to solve all unconstrained problems. The 4 and 14 vertex problems were problems in which all vertices had identical sizes. Note that, I, the number of unconstrained problems solved is quite

n	T1	T2	T3	F	I	μ
4	.002	.01	.012	6	6	2
8	.004	.47	.474	3	58	10
14	1.54	1.88	3.42	1326	1326	4
25	.02	2.42	2.44	5	163	10
46	.28	4.62	4.80	115	182	20

Table 4.4

n = number vertices

T1 = total family solution time (sec.)

T2 = total unconstrained solution time (sec.)

T3 = total solution time (sec.)

F = number of families generated

I = total number of unconstrained solutions

μ = block size

large for the 14 vertex problems, since an unconstrained problem must be solved for each family.

The number of iterations, solutions to unconstrained problems, was fairly constant when no gaps were present. The average was 14 iterations per solution for a given λ_2. With the 46 vertex problem a gap was found after 31 vertices had been removed from the graph by application of the generalized multiplier technique. This resulted in 87 families being generated for the next optimum partition of the graph which at that point consisted of 15 vertices. This was the largest number of families generated for a partition when the sizes of the vertices were not all identical. Generally when gaps did occur, for problems where vertex sizes are mixed, they occurred after a number of vertices had been removed. This meant that the number of families generated was not prohibitive.

Chapter 5

5. Comparison of Program Partitioning Strategies

This section deals with the question of accuracy; that is how close to the optimum is a program partition which is obtained by a sequential selection procedure. We attempt to assess sequential selection as a partitioning strategy by solving a set of problems by two alternative partitioning procedures and comparing the results.

5.1 Alternative Partitioning Strategies

In appendix II a detailed presentation of the unit merge algorithm developed at Informatics [20] is found along with the dynamic programming segmentation algorithm of Kernighan [24]. These algorithms represent different approaches to the problem of graphical partitioning or program reorganization. Both procedures were much faster than the parametric sequential selection procedure, therefore the basic comparison of the algorithms is with accuracy.

5.1.1 Unit Merge Algorithm

Basically, the unit merge algorithm is a procedure which fits into the class of "greedy" algorithms. That is, blocks of vertices are found by selection of the sets of vertices which have the highest weight edges connecting them. Suppose the edges are ordered as follows,

$\{c^1_{ij}, c^2_{\ell k}, \ldots, c^m_{pq}\}$ where,

$c^1_{ij} \geq c^2_{\ell k} \geq \ldots \geq c^m_{pq} \geq 0.$

Now if v_i and v_j, the vertices which are connected by c_{ij}^1, have a feasible total size, i.e., $a_i + a_j \leq \mu$, where a_i and a_j are the sizes of v_i and v_j, they are said to form a feasible merger and hence they are combined to make a new vertex v_i', where v_i' has a size equal to the sum of the sizes of v_i and v_j. All edges previously connected to either of these two vertices are now connected to vertex v_i'.

However, if it is the case that $a_i + a_j > \mu$, then the vertices connected by c_{ij}^2 are considered for a merger. The algorithm continues to test for possible merges in the order of decreasing edge weight. The procedure is terminated when it is no longer possible to make any feasible merger.

The sum of the edge weights between vertices at the time of termination of the algorithm represents the interblock weight of the partition.

5.1.2 Segmentation Algorithms

Kral [28] and Kernighan [24] have independently developed program partitioning algorithms which operate as follows. A program is considered to be a set of n vertices, where the vertices are program units indexed by $\{1, 2, \ldots, n\}$. The size of the vertices are known and the ordering of the vertices is fixed. Minimization of transitions between pages is accomplished by identifying a set of "page breaks" or "break points" between vertices. The vertices between successive break points form the blocks of the partition. This formulation is somewhat simpler than

that of the general program partitioning problem and in appendix II we give the dynamic programming algorithm of Kernighan for its solution.

5.2 Comparative Results

Table 5 summarizes the results after the three algorithms were applied to a collection of graphs. The column labeled A1, A2, and A3 are the resulting interblocks weights obtained respectively from the sequential selection procedure, the unit merge algorithm and the segmentation algorithm. The number of vertices and edges on each graph is represented by N and M respectively. All algorithms were programmed on the IBM 360 model 67. For all graphs the maximum total time required by either the merge algorithm or the segmentation algorithm was 1.6 seconds. The typical times for the algorithm (A1) were given in section 4.6.

N	M	A1	A2	A3
4	4	110	180	110
8	10	1.6	1.6	2.51
14	17	43.5	44.0	43.5
25	50	214.17	204.35	265.30
25	63	218.84	218.97	285.5
46	75	311.2	287.2	386.61
46	95	319.43	339.12	403.2
50	106	141.2	159.1	225.3

Interblock Weights

Table 5

Table 5 shows that the sequential selection algorithm (A1) and (A2) clearly give the best results (smaller interblock weights) in most cases. For the small graphs, $n = 4, 8, 14$, the values obtained by the sequential algorithm (A1) were optimum. The segmentation algorithm (A3) gave results which were in general not as good as the first two algorithms. In some cases a better (smaller) interblock weight was obtained with (A3) by renumbering the vertices.

For the graphs with $n > 14$, which were generated randomly, the optimum results are not known ; here we find for the same value of N that if the graph has a larger number of edges M, we get slightly better results using sequential selection. On the other hand when the graph is more sparse, fewer edges, (A2) gives better results.

The time required by (A1) is considerably more than that required by (A2). For example, the interblock weight obtained by (A1) for $N = 25$ and $M = 46$ is approximately the same as that obtained by (A2). But for this case the total time required by (A1) is 6.4 seconds as compared to .22 seconds for (A2).

For the largest graphs $N = 50$ we obtained the best results when problem II (quadratic knapsack section 4.3) was used as the vehicle for sequential block selection. This result is shown in table 5; the total time required was 13.5 seconds as compared to 1.6 seconds for (A2).

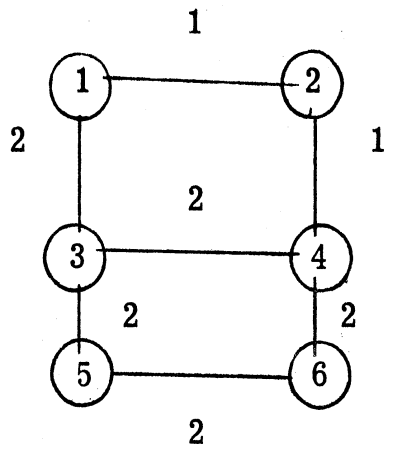
For incorporation of a program packing procedure in a compiler, it is obvious that a fast algorithm such as the unit merge algorithm would be required, since in this case the time added to compilation must be less than the time saved during paging. A procedure such as unit merge meets this requirement.

5. 3 Extensions of Sequential Solution Procedures

In this section we present an expanded branch and bound algorithm for graphical partitioning. Using this expanded algorithm we attempt to deal with some questions such as multiple solutions which may arise when using the sequential selection procedure previously described and the fact that the order in which the partition blocks are selected may affect the overall result.

5. 3. 1 Order of Selected Blocks

Using the sequential optimization procedure to select the block which minimizes the weight of the edges between the remaining graph and the chosen block (or equivalently maximizes the sum total weight of the edges which have both ends adjacent to vertices on the selected block or both ends adjacent to vertices on the graph which remains after the vertices of the selected block are removed) we can in some cases select suboptimal partitions. For example consider the following graph where all vertices have size 1 and the block size is 3.

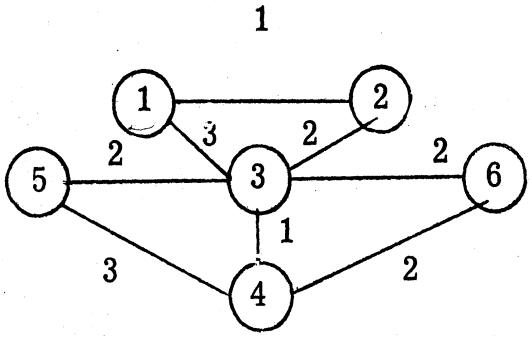


Clearly the feasible block of vertices to remove for minimum interblock weight is $\{1, 2\}$. In this case the sequential selection procedure has MINVER > 1 hence the single block $\{2\}$ is not feasible. Now this gives an incremental cost of 3. The graph which remains can be optimally partitioned for cost of 4; then the total interblock weight is 7. Now it is clear that $\{1, 3, 5\}$ is a block with an interblock weight of 5; this incremental interblock weight is the total interblock weight since no further partitioning is required. This example shows how the sequential selection procedure which selects the optimum feasible block at each stage can be lead to suboptimal results since this procedure has no "look ahead." The branch and bound algorithm which we shall presently describe is one method of dealing with this problem. Next we show how multiple optimum solutions can lead to suboptimal results.

5.3.2 Multiple Solutions

A problem which can be encountered when solving the sequential optimization problem is that of multiple solutions. That is there may be

more than one optimum block which can be selected by sequential optimization algorithm. Consider the following



Here the maximum allowable block size is 3; and all vertices have size 1, with the exception of vertices 4 and 5 which have a size of 2. In this case we have two blocks which may be selected as an initial block in the sequential optimization procedure. We may select either $\{1, 2, 3\}$ or block $\{1, 2\}$ for an interblock weight of 5. It is clear that the additional partitioning cost will differ depending upon which of the initial optimal blocks is selected, e. g. , if $\{1, 2, 3\}$ is selected we can get an optimal partition with interblock weight of 8.

These examples illustrate the fact that it is not possible to characterize a given block as belonging to an optimal partitioning until the complete structure of the partitioning is known. Hence we see that in general any sequential procedure for selection of partition blocks may lead to suboptimal results when the number of blocks required is more than two.

In this next section we describe a procedure for exploring optimal solutions which, at the expense of more computation time, allows one to "look a head" in the sequential selection procedure.

5.4 Branch and Bound Algorighm for Partitioning

One method for dealing with the problem of "order of selection" and "multiple solutions" is to add "look ahead" to the sequential selection procedures. This was done as follows. Suppose we solve the partitioning problem and we have selected m blocks of a partition V_1, $V_2, \ldots V_m$. And suppose block V_1 has size s_1, where $s_1 \leq \mu$ where μ is the bound on the block size used in the sequential selection procedure. We have discussed two particular ways of selecting blocks (problems II and III in section 4.5). It was found that in general the second strategy (problem III) gave the best results with the algorithm discussed in chapter 4. However with the branch and bound strategy we used the quadratic knapsack (problem II)as the tool for block selection.

Associated with the partition $V_1, V_2, \ldots, V_m$ is an interblock weight TMIN; we now want to discover if we had selected another block smaller than V_1 as our initial partition block whether we could have obtained an overall smaller interblock weight TMIN. Since we know that the block with maximum weight and largest size obtained from the original graph has size s_1, originally, we set the bound in the block size μ at $\mu' = s_1 - q$, where $q > 0$ and start the problem over again.

With this particular μ the original initial block will not be feasible, hence the algorithm will select a new first block V_1^{ℓ}. Now the graph which remains to be partitioned is not the same graph as when originally we selected the first block using μ as a size bound. In figure 5.1 we see a representation of the branch and bound scheme. Each vertex represents the state of the graph to be partitioned. G_1 is the original graph. The arcs labeled s_1 leading to the node labeled G_2 means that after selecting the first block of size s_1 we obtained the graph G_2. At each vertex G_i there is an associated cost T_i for obtaining the set of blocks leading to the graph G_i. We can compute the incremental cost IC of going from G_i to G_{i+1} using s_k, $k = 1, 2, 3$ for the graph shown, and if $T_i + IC > TMIN$ we stop the search along this path. Otherwise we continued until we got a complete partition. If at the end we obtained a smaller TMIN we retained this new partition and this cost is our new TMIN. We then back tracked to the last vertex at which we had not considered alternative block size.

The second phase of the algorithm for unconstrained optimization was programmed so as to record multiple solutions. It was then possible to utilize the same branch and bound routine to trace for alternative partitions which arise due to multiple maximizing vector found in the second phase. However, in this case s_1, s_2, etc. represent different blocks selected due to different maximizing vectors, not different block sizes.

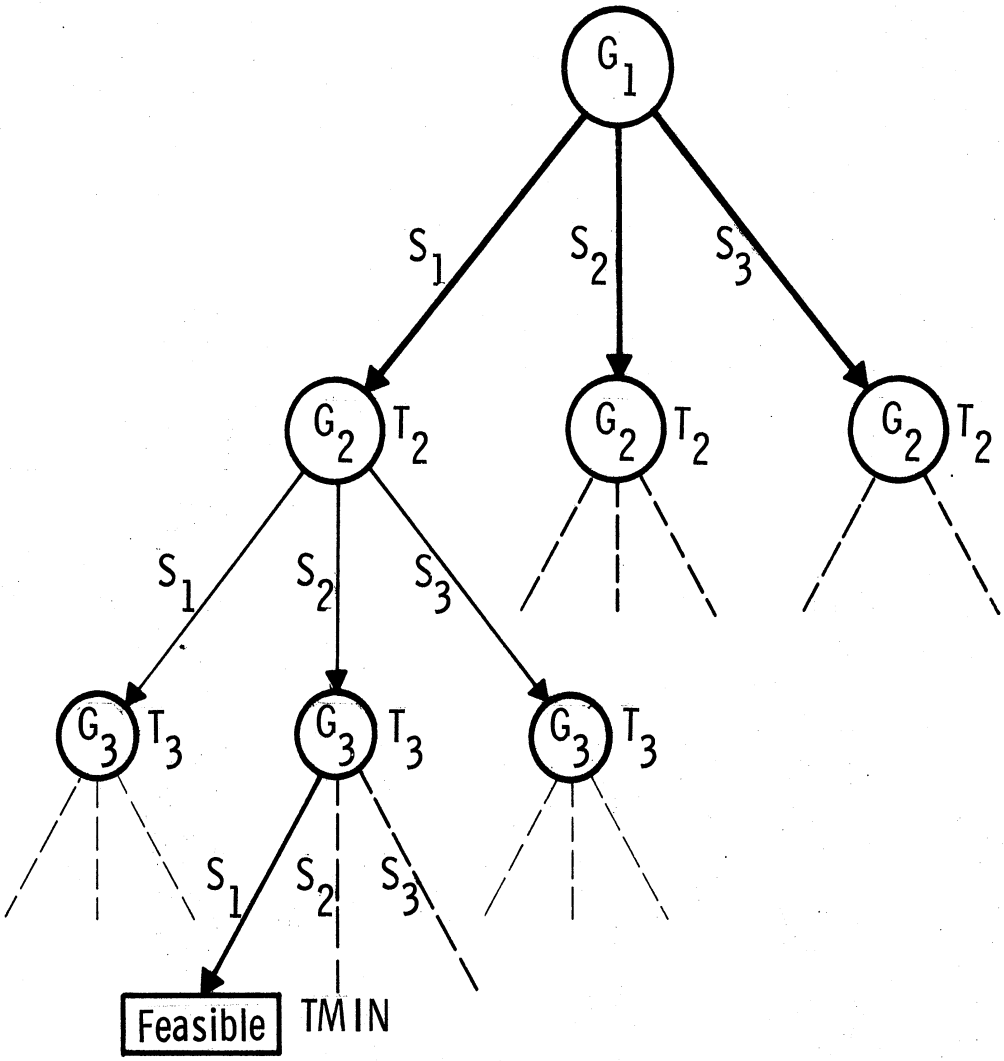

G1
S1
S2
S3
G2 T2
G2 T2
G2 T2
S1
S2
S3
G3 T3
G3 T3
G3 T3
S1
S2
S3
Feasible
TMIN

Figure 5.1

In using the expanded branch and bound procedure it was found that the computational time grew quite rapidly if there were many branches to trace in the tree of figure 5.1. The increase in computation time did not seen to justify this approach to graphical partitioning. The use of $T_i + IC \leq TMIN$ as a test was not sufficiently powerful to avoid lengthy tracing through the tree of figures 5.1.

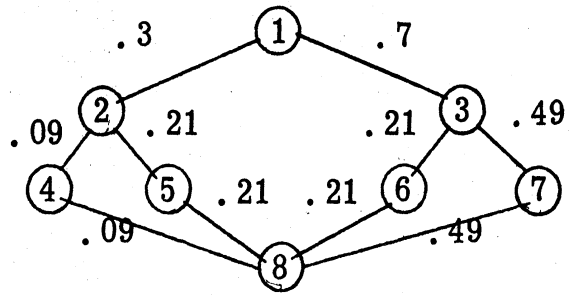
If one were to pursue this particular application, a way of decreasing the tree searching is to use a stronger test to decide if a particular path is to be followed. That is, calculate a lower bound on the additional cost before a particular path in the tree is taken. For example suppose we have reached mode G_i which represents the graph after i - 1 block have been removed. We could use the max - flow/min cut algorithm of Gomory and Hu [16] to calculate the minimum additional cost to partition G_i, since this algorithm gives the minimum cost to partition G_i into two blocks independent of the block size. This of course, could add more time to the algorithm so that the trade-off between additional computation at each vertex G_i and the total number of paths searched would have to be investigated.

5.5 Summary

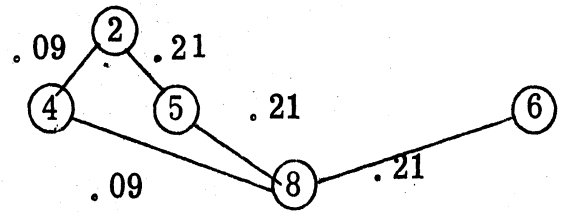
In using the different algorithms for partitioning, it became clear that the partitions obtained by the sequential selection algorithm are dependent upon the "uniqueness" of the blocks selected in the early

stages of the procedure. This means, of course, that the order in which blocks of an optimum partition are selected is very important. This is why for partitions where the number of blocks is limited, and hence order of selection is not as significant, the sequential algorithm gave optimal results. In considering other algorithms which select blocks sequentially, we find this same problem. For example the Kernighan - Lin [25] algorithm is a sequential selection procedure which divides the graph into two subgraphs of equal vertices at each step, it attemps to minimize the sum of the weights of edges between the two subgraphs. But even here, where no attempt is made to get feasible size blocks, there is no way of choosing between alternative optimum "two-way" partitions in the early stages of their procedure. This means that after the first optimum two way partition is found, half of the vertices of the original graph can never be considered to be on the same block of a partition.

On the other hand, when we consider the unit merge algorithm, we find that the "merges" made do not tend to disconnect or rule out particular sets of vertices for blocks as rapidly as sequential selection. The major problem with the unit merge procedure is that it is highly dependent on the "order" in which the maximum weight edges are selected. Consider the example which follows in which all vertices have unit size and the block size is 3.



This example was given in [20] and the optimal partition was said to have an interblock weight of 1.3. Vertices 1, 3 and 7 are first selected by the merge algorithm for the same block. This leaves the following graph.



Now it is possible using the merge algorithm to select vertices 2, 5 and 8 as the next partition block. This will result in an interpage weight of 1.39 which is not optimal. This problem is always encountered when the merge algorithm is used with graphs where some of the edges have identical weights.

Segmentation procedures [24], [28] can be done efficiently with dynamic programming techniques. The computation time for these algorithms are competetive with the merge algorithm. With the examples we considered, the partitions found by segmenting or finding break points

were in general the worst of the three partitionings. In some cases we could find a better partition by renumbering the vertices. It appears that for graphs in which several vertices have a high degree, i.e., a large number of incident edges, this procedure will give partitions which have a relatively high interblock weight. However this approach to partitioning was designed to be used on the instruction level. That is the program unit which the vertices represent are individual instructions rather than sets of instructions. A graph representing programs at this level would tend to have in general a low average vertex degree. For program representation at this level segmentation algorithms are applicable. However, one is immediately confronted with the problem of gathering statistics for programs at this level of detail.

Summarizing we saw that sequential selection tended to give slightly better partitions than the greedy algorithm for graphs with a high edge to vertex ratio. There is however a trade-off in the increased computational time needed for the sequential selection procedure and in most cases the additional accuracy obtained by this procedure would not be justified in terms of additional time. For graphs which are sparsely connected, the merge algorithm gave results which make it preferable.

Segmentation procedures should be used only in graphs for which the average degree of the vertices is relatively small, i.e., at instruction level.

In the next chapter we give an example of the changes in paging performance for a system translator which has been reorganized after application of a program partitioning algorithm.

Chapter 6

6. Application to Systems Programming

In this chapter we assess the impact of program partitioning on a systems program's performance in a real paging environment. A program was written to take the results of an optimal grouping of a set of subroutines of a systems program and reorganize the set of routines such that the interpage references among the routines were reduced. The program graph for the systems program (SNOBOL 4) was sparsely connected; therefore the Merge algorithm was used for partitioning.

Due to its modular structure we found for purposes of this test that the SNOBOL 4 translator on the Michigan MTS system was well suited as a testing vehicle. Performance results for both the normal and the reorganized version of the translator are given.

6.1 Paging of a SNOBOL Translator in MTS

General computing services at The University of Michigan are provided on the IBM 360 model 67. The software system is called MTS, i.e., the "Michigan Terminal System". The Michigan Terminal System is a reentrant job program which runs under the University of Michigan Multi-Programming System (UMMPS). In addition to batch processing, MTS has the capability of controlling and executing programs from remote terminals. Programs run under MTS are paged in a multi-programmed environment.

For purposes of assessing the impact of programmed reorganization on paging performance, a SNOBOL 4 translator was analyzed and optimized using the procedures previously discussed. A special version of the translator was assembled which allowed determination of the following,

1. the inter-page branching between procedures,
2. the number of times each procedure is called,
3. the time spent in each procedure,
4. the average time per call for each procedure.

The MTS data collection facility [37] was used to collect the statistical data from the special version of the translator. The data collection facility produced a tape with all the relevant information from which the appropriate data could be derived. This data was used to construct a program graph of the SNOBOL 4 translator.

The SNOBOL 4 translator used for this test consisted of 106 independent procedures which call on each other during execution of the program. The procedures ranged in size from 30 bytes to 3076 bytes, with the total translator having 18 pages of procedures. A page consists of 4096 bytes on the IBM 360.

To maintain a constant environment this paging experiment was run in the early (4:00AM, 19 Dec. 1970) hours of the morning with few other users on the system. Two special statistical versions of SNOBOL were assembled, one reorganized and one left in its normal state. A set of five SNOBOL programs were run using each translator. To

create a constant system paging environment each translator was run with a special background job call PAGE-IT. The PAGE-IT job obtains a large number of virtual storage pages and references them cyclically in rapid succession. PAGE-IT was designed such that when run with a moderate load of normal MTS tasks, the drum channel programs are kept running continuously. The effect of this is to force pages of other programs out of storage at a faster than normal rate.

6.2 Reorganization of a SNOBOL Translator

The SNOBOL translator or program functions as an interpreter; that is its operation is divided into two phases, translation and execution. The translation phase consists of an examination of the source language program (problem program) and a determination of the proper subroutine to execute the tasks specified by the source statement. Execution is that phase of the SNOBOL program in which control is given to certain SNOBOL subroutines for execution of tasks determined during translation.

As cited earlier the SNOBOL translator consisted of 18 pages of code. There are an additional 30 pages of data. The reorganization performed consisted only of reorganization of the pages of code since it was not possible to affect the organization of the data pages. During the translation phase of the SNOBOL translator only two data pages are referenced. Therefore it was assumed that the behavior during this phase of the translator could best reflect the paging results, due to reorganization, of the routines of the translator.

Program	S1	S2	D
1	7560	7324	236
2	7124	7029	95
3	11332	11242	90
4	4348	4365	-17
5	3209	3063	146

P=4082 during S1
P=4397 during S2

Table 6.1

Program	S1	S2	D
1	3771	3739	32
2	3433	3612	-179
3	5496	5495	1
4	2186	2180	6
5	1545	1566	-21

P=182 during S1
P=831 during S2

Table 6.2

Table 6.1 and Table 6.2 represent the translation times of the five test problem programs used in our paging experiment. The column labeled S1 represents the SNOBOL translation time of the problem programs which used the translator which was not reorganized; S2 represents the translation times of the test programs using the reorganized version of SNOBOL; and D is the difference S1-S2; all times are in milliseconds. Table 6.1 summarizes the results taken during operation of the special program PAGE-IT. The value P beside the tables represents the number of drum reads recorded during operation of these programs; their number is indicative of the paging activity occurring during the running of the test programs. The translation times include time taken for paging.

First we note that the times in Table 6.1 are about twice as large as those in Table 6.2; this is due of course to the additional time for paging. For example when the five problem programs were run with the translator designated S1, there were 4082 drum reads during the execution of these programs with PAGE-IT creating a heavy load. When these same programs were run on S1 without the additional paging load there were only 182 drum reads.

For the first problem program we see that when there is relatively small paging load (Table 6.2) the difference in translation time is 32 milliseconds with the reorganized translator S2 being faster. However, with this same program under heavy paging conditions (Table 6.1) we get a translation time difference of 236 milliseconds,

with S2 being faster. For this problem program we get a translation time difference of approximately 1%, i.e., the time for S1 is approximately 1.01 times the time for S2, under a small paging load. The translation time difference is about 3% when there is much paging activity with the reorganized version faster.

With problem programs 3 and 4 there is little difference in translation times under a light paging load. However when paging is increased the increase in translation time for problem 3 is slightly less than 1% with S2 the faster version. With problem 4 the reorganized version S2 was the slower with a heavy paging load. The explanation for this is not clear except that the reorganization of the SNOBAL translator was based on its "average" behavior; therefore it could be expected to run slower for some types of problem programs.

Problem program numbers 2 and 5 had relatively high time differences under the small paging load. In these cases it was the reorganized version which was slower with little paging (Table 6.2). But when the paging load is increased for both these problem programs the reorganized version was faster by at least 1.5%.

In conclusion we note that in 4 of the 5 problem programs superior performance, under heavy paging, was observed by the reorganized SNOBOL translator. In the one case, problem program 4, where the reorganized program was slower the difference was less than 1/2%.

The data for which the program graph for SNOBAL was constructed was obtained by running a program which used a wide mix of the translator's facilities. Therefore when the problem programs, not the same program from which the program graph was constructed, were run a variation could be expected since all problem programs use different parts of the translator with different frequencies.

The portion of the SNOBAL translator used for data is larger than that used for code (30 pages versus 18 pages). Therefore the paging due to data referencing is significant. We could not affect the organization of the data pages. The original version S1 was better organized with respect to data referencing. When the translator was reorganized the paging due to data referencing actually increased, i.e., the value of P is always higher for the reorganized translator.

Although the statistics were taken at a time in which system use was minimal, we could not completely control the system environment. Users could sign on and thereby affect the overall system paging strategy and thereby the statistics we collected.

A conclusion which may be drawn from our results is that the paging efficiency can be increased by a reorganization of the code of a program. Even though the efficiency increase may be relatively small, the fact that the systems routines are heavily used can result in significant savings in paging overhead. If little data referencing is done this is all that is necessary. But if many data pages are used

code reorganization should not take place without consideration of the references to data pages.

Chapter 7

7. Summary and Future Research

This section summarizes the results of this study and suggests areas where the research may be extended.

The central purpose of this research has been the development of an algorithm for finding specific optimal groupings. Towards this end we have done the following:

(a) developed an algorithm for solution of a quadratic binary program of large dimension;

(b) investigated one specific application of optimal groupings to computer programming;

(c) made computer comparisons of algorithms for computer program partitioning;

(d) restructured a systems program and monitored paging behavior.

We have developed an algorithm which has application to a variety of problems. For the specific application of computer program partitioning we found the unit merge algorithm to give the best performance in terms of running time and accuracy; for some graphs the Unit Merge algorithm could give results which were inferior to those given by sequential selection. The speed and simplicity of the Unit Merge algorithm makes it attractive as a procedure to incorporate in a compiler. Segmentation procedures

were found to be effective only for graphs of low average vertex degree.

A system program was reorganized and the paging behavior due to this reorganization was measured. It was found that considerable paging overhead could be decreased by program reorganization. For programs with large amounts of data it was found that data pages must be considered in any scheme of overall program reorganization.

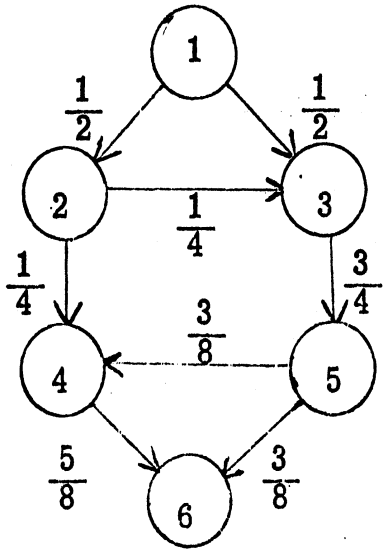
In the next section we consider some extensions to the problem we have discussed.

7.1 Extensions to Quadratic Algorithm

We have discussed how generalized multipliers maybe used to solve the quadratic problem for $m \leq 2$. There should be more research in extending the generalized multiplier approach to large values of m.

7.2 Program Partitioning with Paging

In the previous work, we have not made use of the fact that a program may have more than a single page in main memory at a given instant. When this aspect of memory management is considered, one can alter the nature of a given program partitioning. Consider the following example.



For this graph all vertices except 1 and 6 have size 1, vertices 1 and 6 have size 2; the page size is two. The unique optimum partition for the above program graph is

Page 1 1

Page 2 2, 4

Page 3 3, 5

Page 4 6

This partition corresponds to an interpage transition weight of 2 1/8 which is minimum. However if we assume that there is a non-zero probability p(k) that k pages of this program are in main memory simultaneously, where k=1, 2, 3, 4 and $\sum p(k) = 1$, then we see that an optimum partition will depend upon the distribution p(k). For example suppose p(k) = 0, k=1, 3, 4, and p(2) = 1. This means that we can have pages 1 and 2 or pages 3 and 4 in core simultaneously. When we are allowed this configuration our "interpage" cost is now only 1 3/4; since there is no cost for transfer between vertices 1, 2, and 4 or 3, 5, and 6. But we notice that we can reorganize this program parition such that our interpage cost is 1 if we define the new packing as,

Page 1 1

Page 2 2, 3

Page 3 4, 5

Page 4 6

With the new formulation pages 1 and 2 and pages 3 and 4 are simultaneously core resident. The point is that the optimum partition found under a "static" consideration is not necessarily optimum when one allows multiple pages to be core resident.

When one considers multiple pages in core, a natural question is, how partitioning of programs interacts with "page turning" or page placement/replacement algorithms. One can formulate the partitioning problem in such a manner that one has included an implicit page turning algorithm. Suppose we have distribution p(k), the probability that a given program has k pages resident. We then would like to select a partitioning which somehow takes advantage of the information given by distribution p(k).

Let us define $x_{ij}^{1} = 0$ if vertices i and j are on the same page,

$= 1$ otherwise.

For $k > 1$ define $x_{ij}^{k} = 1$ if when control is at vertex i vertex j is not on any of the k pages which are core resident,

$= 0$ otherwise.

Now a cost function which takes into consideration the fact that there is not necessarily a cost associated with interpage transfers is

$$(1)\quad F = \sum'_{i,j,k} c_{ij} x_{ij}^{(k)}\, p(k).$$

In this case we get a contribution to the cost function **only** if vertices are on different pages which do not occupy core simultaneously. The distribution p(k) "weights" the cost according to the statistics on core usage. The constraints for such a formulation could be as follows. To assure that the vertices placed on a page don't exceed the capacity of a page we have,

$$(2)\quad \sum'_{j} (1 - x_{ij}^{k}) a_i < k\mu, \text{ for all } i, k;$$

where a_i is size of vertex i and μ is page size. A solution is considered feasible if and only if for $x_{ij}^{k} = 0$ and $x_{ij}^{1} = 1$ then $x_{ij}^{p} = 0$ for all $p > k$. This constraint simply says that if vertices i and j are on pages A and B respectively for k pages in core and control at A, then if more pages are allowed in core with control at i, we will consider only those solutions which still consider B to be among the resident pages. This can be stated as

$$x_{ij}^{t} - x_{ij}^{k} \geq 0 \text{ if } t \leq k, \text{ for } t = 1, 2, \ldots m,$$

Now x_{ij}^{k} says whether j is on any page among the k pages which are resident when control is at i; this should be the same for all other vertices p which are on the same page as i, i.e., if $x_{ij}^{k} = 0$ and $x_{ip}^{1} = 0$ then $x_{pj}^{k} = 0$. We state this as

$$(3)\quad \sum_{i \neq j} (1 - x_{ij}^{1}) \sum_{k=1}^{m}{}' \sum_{p=1}^{n} \left[(x_{pi}^{k} - x_{pj}^{k})^2 + (x_{ip}^{k} - x_{jp}^{k})^2 \right] = 0.$$

for all i, j.

And of course $x_{ij}^{k} \epsilon \{0, 1\}$ for all i, j, k and $x_{ii}^{k} = 0$ for all i, k. Equation (1-3) defines an optimum problem in boolean variables which is more general than the problem defined in section 2. The information $\{x_{ij}^{k}, k=1, 2, \ldots, m\}$ obtained in the solution can be used in a page turning policy in order to minimize the number of page interrupts.

The optimization problem formulated for multiple pages core resident is indeed very difficult to solve for cases of practical interest. A possible area of future research in the formulation of more tractable optimization models which retain the features of page multiplicity.

Implicit with a model which considers multiple pages in core is the consideration of page fetch and replacement strategies. More research into this area is appropriate.

In the models developed no consideration has been given to multiprocessing program parallelism. Models which incorparate paging could include these features.

Appendix I

Solution of Linear Pseudo-Inequalities

In this section we present a branch and bound algorithm due to Hammer and Rudeanu [19] for solution of a linear pseudo-Boolean inequalities of the following form

$$c_1x_1 + c_2x_2 + \ldots + c_nk_n \geq d, \tag{1}$$

where $c_1, c_2, \ldots, c_n$, d, $(c_1 \geq c_2 \geq, \ldots, \geq c_n > 0)$ are known real constants and x_i (i=1, . . . , n) are Boolean unknowns.

Any Boolean inequality can be put into standard form (1) by application of the following transformation. Let

$$a_1z_1 + b_1\bar{z}_1 + \ldots + a_nz_n + b_n\bar{z}_n \geq k \tag{2}$$

be the general form a linear psuedo-Boolean inequality takes where a_i, b_i, (i=1, . . . , n) and k are known constants and z_i (i=1, . . . , n) and $\bar{z}_i = 1 - z_i$ are the Boolean unknowns. We can assume, with no loss of generality, that $a_i \neq b_i$ for all i; also the proper sense of the inequality is always obtained by appropriate multiplication by -1.

For each i, let us set

$$x_i = \begin{cases} z_i, & \text{if } a_i > b_i \\ \\ \bar{z}_i, & \text{if } a_i < b_i \end{cases} \tag{3}$$

Now the term $a_iz_i + b_i\bar{z}_i$ becomes

$$a_i z_i + b_i \bar{z}_i = \begin{cases} (a_i - b_i)x_i + b_i, & \text{if } a_i > b_i \\ (b_i - a_i)x_i + a_i, & \text{if } a_i < b_i \end{cases} \tag{3a}$$

Thus equation (2) becomes

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \geq d$$

where $c_1, c_2, \ldots, c_n, d$ are all known constants. By reindexing we have $c_1 \geq c_2 \geq \ldots \geq c_n > 0$.

Throughout we will confine our attention to the solution of inequalities of standard form.

Following Hammer and Rudeanu we need the following definitions.

Definition 1.

Let $S = (x^*_1, \ldots, x^*_n)$ be a solution of (1) and let I be a set of indices $I \subseteq \{1, 2, \ldots, n\}$. Let $\Gamma(S, I)$ be the set of all Boolean vectors $(x_1, \ldots, x_n)$ satisfying

$$x_i = x^*_i \quad \text{for all } i \in I,$$

The other variables x_i, $i \notin I$ are arbitrary. If all Boolean vectors $(x_1, \ldots, x_n) \in \Gamma(S, I)$ satisfy (1), then $\Gamma(S, I)$ is said to be a family of solutions to (1). The pair (S, I) is said to generate the family $\Gamma(S, I)$; the variables x_i for which $i \in I$ are called the fixed variables of the family. If $I = \{1, 2, \ldots, n\}$ the family consists of the single vector $(x^*_1, \ldots, x^*_n)$ and is said to be degenerate.

Definition 2.

A vector $(x^*_1, \ldots, x^*_n)$ satisfying inequality (1) is called a basic solution of (1), if for each index i such that $x^*_i = 1$, the vector $(x^*_1, \ldots, x^*_{i-1}, 0, x^*_{i+1}, \ldots, x^*_n)$ is not a solution of (1).

For solution of (1) the following three lemmas are needed.

Lemma 1

Let $(x^*_1, \ldots, x^*_p, x^*_{p+1}, \ldots, x^*_n)$ be a basic solution of (1), then $(x^*_{p+1}, \ldots, x^*_n)$ is a basic solution of the inequality

$$\sum_{j=p+1}^{n} c_j x_j \geq d - \sum_{k=1}^{p} c_k x^*_k$$

Lemma 2

If $(x^*_{p+1}, \ldots, x^*_n)$ is a basic solution of the inequality

$$\sum_{j=p+1}^{n} c_j x_j \geq d,$$

then $(0, 0, \ldots, 0, x^*_{p+1}, \ldots, x^*_n)$ is a basic solution of (1).

Lemma 3

If $d > 0$ and $(x^*_2, \ldots, x^*_n)$ is a basic solution of

$$\sum_{j=2}^{n} c_j x_j \geq d - c_1, \tag{4}$$

then $(1, x^*_2, \ldots, x^*_n)$ is a basic solution of (1).

Proof:

If $x^*_2 = \ldots = x^*_n = 0$, the lemma follows immediately; therefore assume that $\sum_{j=2}^{n} x^*_j > 0$. It is obvious that $(1, x^*_2, \ldots, x^*_n)$ is a solution of (1); hence it remains only to show that it is a basic one.

By hypothesis $(x^*_2, \ldots, x^*_n)$ is a basic solution of (4), therefore it follows that for every k $(2 \leq k \leq n)$ such that $x^*_k = 1$

$$\sum_{\substack{j=2 \\ j\neq k}}^{n} c_j x^*_j < d\text{-}c_1$$

Hence we have that

$$c_1 + \sum_{\substack{j=2 \\ j\neq k}}^{n} c_j x^*_j < d \qquad \text{or}$$

$(1, x^*_2, \ldots, x^*_{k-1}, 0, x^*_{k+1}, \ldots, x^*_n)$ does not satisfy (1) or $(1, x^*_2, \ldots, x^*_n)$ is a basic solution of (1). Further

$$\sum_{j=2}^{n} c_j x^*_j = c_k + \sum_{\substack{j=2 \\ j\neq k}}^{n} c_j x^*_j < c_k + d\text{-}c_1 \leq d,$$

therefore $(0, x^*_2, \ldots, x^*_n)$ is not a solution of (1). This completes the proof of lemma 3.

All basic solutions are found by repeated application of the above three lemmas. Table A1 summarizes those results.

Table A1

Condition	Conclusion	Justification
$d \leq 0$	Unique basic solution $x_1 = x_2 = \ldots = x_n = 0$	obvious
$d > 0$ and $c_1 \geq \ldots \geq c_p > d >$ $c_{p+1} \geq \ldots \geq c_n$	a. For each $k=1, 2, \ldots, p$ $x_k = 1$ $x_1 = x_2 = \ldots x_{k-1} = x_{k+1} = \ldots = x_n = 0$ is a basic solution b. If any other basic solutions exist, they are characterized by the property: $x_1 = x_2 = \ldots = x_p = 0$, and $(x_{p+1}, \ldots, x_n)$ is a basic solution of $\sum_{j=p+1}^{n} c_j x_j \geq d$	obvious by lemmas 1, 2
$d > 0$, and $\sum_{i=1}^{n} c_i < d$	No solution	obvious
$d > 0$, $c_1 < 0$ $\sum_{i=1}^{n} c_i = d$	Unique basic solution $x_1 = x_2 = \ldots = x_n = 1$	obvious
. . .	. . .	. . .

Table A1 (continued)

Condition	Conclusion	Justification
$d > 0$, $c_1 < d$ $\sum_{i=1}^{n} c_i > d$ and $\sum_{i=2}^{n} c_i < d$	The basic solutions (if any) are characterized by the property: $x_1=1$, and $(x_2, \ldots, x_n)$ is a basic solution of $\sum_{j=2}^{n} c_j x_j \geq d-c_1$	by lemmas 1, 3
$d > 0$, $c_1 < d$ $\sum_{i=1}^{n} c_i > d$ and $\sum_{i=2}^{n} c_i \geq d$	The basic solutions (if any) are characterized by the property: either $x_1=1$ and $(x_2, \ldots, x_n)$ is a basic solution of $\sum_{j=2}^{n} c_j x_j \geq d-c_1$, or: $x_1=0$ and $(x_2, \ldots, x_n)$ is a basic solution of $\sum_{j=2}^{n} c_j x_j \geq d$	by lemmas 1, 3, 2

Each basic solution, S, can be associated with a family of solutions $\Gamma(S, J_s)$ as follows. Let i_0 be the last index for which $x^*_i = 1$, (i.e., $x^*_{i_0} = 1$, and $x^*_i = 0$ for all $i > i_0$), and let J_s be the set of all indices $i \leq i_0$. Then $\Gamma(S, J_s)$ is the set of all vectors $(x_1, \ldots, x_n)$ satisfying

$$x_i = \begin{cases} x^*_i & i \leq i_0 \\ \text{arbitrary} & i > i_0 \end{cases}$$

It can be shown (see [18]) that if $S_1, \ldots, S_m$ are the basic solutions of (1), and $\Gamma(S_k, J_k)$, $k=1, \ldots, m$ the associated families, then every solution $(x_1, \ldots, x_n)$ of (1) belongs to exactly one family. We now present an example.

Example

$$2z_1 + 6z_2 - 4z_3 + z_4 + 3z_5 \leq 6$$

Rewriting as

$$-2z_1 - 6z_2 + 4z_3 - z_4 - 3z_5 \geq -6$$

$$y_1 = \bar{z}_1,\ y_2 = \bar{z}_2,\ y_3 = \bar{z}_3,\ y_4 = \bar{z}_4,\ y_5 = \bar{z}_5$$

By applying transformations (see equations (3) and (3a)) we get

$$(2y_1 - 2) + (6y_2 - 6) + 4y_3 + (y_4 - 1) + (3y_5 - 3) \geq -6$$

$$2y_1 + 6y_2 + 4y_3 + y_4 + 3y_5 \geq 6;$$

by reindexing $x_1 = y_2$, $x_2 = y_3$, $x_3 = y_5$, $x_4 = y_1$, $x_5 = y_4$ we get the standard form

$$6x_1 + 4x_2 + 3x_5 + 2x_4 + x_5 \geq 6$$

$x_1=1$, $x_2=x_3=x_4=x_5 = 0$ by case 2 (see table A1).

(a) (1, 0, 0, 0, 0)

So by part (b) of case two $x_1=0$ and one new equation is

$$4x_2 + 3x_3 + 2x_4 + x_5 \geq 6$$

An examination of the coefficients shows that we have case 6; hence $x_2=1$ and we have to find the basic solution of

$$3x_3 + 2x_4 + x_5 \geq 2.$$

which we see in case 2, $x_3=1$, $x_4=0$, $x_5=0$, or

(b) (0, 1, 1, 0, 0)

and $x_3=0$, $x_4=1$, $x_5=0$, or

(c) (0, 1, 0, 1, 0).

Proceeding to part (b) of case 2 we have

$$x_5 \geq 2$$

which leads to case 3.

We here follow this branch as far as possible; we therefore return to the second part of case 6, $x_2=0$

$$3x_3 + 2x_4 + x_5 \geq 6.$$

This is case 4, $x_3 = x_4 = x_5 = 1$,

(d) (0, 0, 1, 1, 1)

This terminates the search for basic solutions. Tables A2 and A3 summarize the results.

Table A2

Solution	x_1	x_2	x_3	x_4	x_5
a	1	0	0	0	0
b	0	1	1	0	0
c	0	1	0	1	0
d	0	0	1	1	1

Table A3

Family	x_1	x_2	x_3	x_4	x_5
$\Gamma(S_1, I_1)$	1	-	-	-	-
$\Gamma(S_2, I_2)$	0	1	1	-	-
$\Gamma(S_3, I_3)$	0	1	0	1	-
$\Gamma(S_4, I_4)$	0	0	1	1	1

The dashes in Table A3 indicate the variables which are arbitrary in a given family, i.e., $I_1 = \{1\}$

$I_2 = \{1, 2, 3\}$, $I_3 = \{1, 2, 3, 4\}$, $I_4 = \{1, 2, 3, 4, 5\}$

so that family $\Gamma(S_4, I_4)$ is degenerate.

After applying the inverse transformation we get Table A4 which represents the families of solutions in terms of our original variables.

Table A4

Family number	z_1	z_2	z_3	z_4	z_5
1	-	0	-	-	-
2	-	1	1	-	0
3	0	1	1	-	1
4	0	1	0	0	0

Appendix II

Two Partitioning Algorithms

In this appendix we give two algorithms for graphical partitioning. The first algorithm is the unit merge procedure developed at Informatics [19]. The second algorithm is the segmentation procedure of Kernighan [24].

II.1 Informatics Unit Merge Algorithm

The unit merge algorithm is suitable for application to any graph G where the edge weight c_{ij} are not all the same. The procedure assumes that the graph G is represented by an upper triangular matrix W,

$$W = [\, w_{ij} \,]$$

$$w_{ij} = c_{ij} + c_{ji} \qquad i < j$$

$$= 0 \qquad i \geq j$$

c_{ij} is the weight of edge between v_i and v_j in graph G.

The algorithm proceeds as follows.

1) The matrix W is searched, and its largest element w_{ij} is located.
2) If the size $a_i + a_j$ (of vertices v_i and v_j) does not exceed μ the block capacity, v_i and v_j are merged. The matrix W is accordingly updated.
3) The process is repeated until all possible mergers have been made. That is, the process terminates when for each $w_{ij} > 0$, $a_i + a_j > \mu$.

4) In order to reduce the number of pages required, any remaining possible mergers are made with no regards to W. The mergers made during this phase of the algorithm do not change the interblock weight. They simply fill out any remaining empty space or partially filled pages.

In order to update the graph when v_j is merged with v_i ($i < j$), the newly formed unit (vertex) is named v_i. The new weight is a_i, the sum of the old a_i and a_j. For all k w_{kj} and w_{jk} become zero. The old values of w_{kj} and/or w_{jk} are added to w_{ki} or w_{ik}.

II.2 Dynamic Programming Algorithm for Segmentation

Kernighan has developed an algorithm for the segmentation of program which requires computations on the order of n^2. Where n is the number of vertices of the graph. We present the model for his optimization procedure.

Let G be a directed graph. A node (vertex) j is a successor of a node i iff (i,j) is an edge in G. In this case, i is a predecessor of j. A program graph is a directed graph G of n + 2 nodes $\{0,1,2,\ldots n,n+1\}$ such that

1. There is no edge (m,0), for m in G; node 0 has no predecessors.
2. There is no edge (n+1,M), for m in G; node n+1 has no successors.
3. There is no edge (m,m), for m in G.

4. G is connected in the sense that for any node m in G, there exists a path from 0 to n+1 which includes node m.

Nodes 0 and n+1 are called the initial node and terminal node of G respectively. The vertices of G have integral weights $\{w(i),\ i=0,1,\ldots n+1\}$ such that $w(0) = w(n+1) = 0$, and $0 < w(i) \leq p$ for all i in (i,n); p is a positive integer representing the pages size. A segmentation of the program graph G is designated as a partition of the nodes of G into k disjoint subsets $G(1),\ldots,G(k)$ such that

1. $\bigcup_{j=1}^{k} G(j) = G$
2. The nodes in any G(j) are contigious; that is G(f) contains nodes $i, i+1,\ldots,m-1,m$.
3. $\sum_{i \epsilon G(j)} w(i) \leq p$; that is the sum of the weights of the nodes in each subset of the partition is less than or equal to p. The subset G(j) is called a page of the segmentation.

Let b(j) be the minimum node number in G(j), that is, the name of the first node of G(j). The number b(j) is called the break point or page break. The set $\{b(j),\ j=1,\ldots,k\}$ uniquely identifies the partition $\{G(j)\}$, $b(1) = 1$.

Each edge (i,j) of G is assigned a nonnegative cost c(i,j) and the cost of a segmentation is degined to be the summation of c(i,j) over all i and j such that i and j are not in the same page. The algorithm below obtains an optimal segmentation, i.e., one with minimum cost. The

distance d(i,j) from node i to node j ($i \leq j$) in a program graph G is defined as $d(i,j) = w(i) + w(i+1) + \ldots + w(j)$; thus the distance is the sum of the weights of all nodes from i to j inclusive.

Optimal Segmentation Algorithm

Let C(x,y) be the incremental cost for a page break at x, given that the previous page break is at y,

$$C(x,y) = \sum_{\substack{y \leq i < x \\ j \geq x}} [\, c(i,j) + c(j,i)]$$

and let T(x) for any x in $\{1,\ldots n\}$ be the minimum partial cost (as far as node x) for any segmentation of a section of a program $\{1,2,\ldots,n\}$, with a break at x. Thus T(1) = 1; T(x) is evaluated iteratively. The algorithm goes as follows.

1. Set T(1) = 0
2. For x from 2 to n+1 in steps of 1, set $T(x) = \min_{y} [\, T(y) + C(x,y)]$ where the minimization is done over all y such that $d(y,x-1) \leq p$. If more than one y satisfies the condition, choose the smallest. Set L(x) = y.
3. Set Total cost = T(n + 1)

   set z(0) = n+1

   set k = 1
4. While $z > 1$, do

   z(k+1) = L [z(k)] ; k = k+1

5. Break points are z(1), z(2),...,z(k), in descending order.

A proof of the optimality of this algorithm can be found in [24].

Appendix III

Graph Theoretic Formulation

In discussion of graphical models of computer programs it is very helpful to present concepts in a graph theoretic context. And it is in this regard that we present the following definitions following Busacker and Saaty [7].

A graph is defined as follows: A graph consists of nonempty set V, a (possibly empty) set E disjoint from V, and a mapping Φ of E into $V \times V$. The elements of V and E are called the vertices and edges of the graph, respectively, and Φ is called the incidence mapping associated with the graph.

An edge $e \in E$ is said to incident with vertices $v, w \in V$ if $\Phi(e) = (v, w)$. The vertices incident with an edge are called its end points, and are also said to be joined by the edge. Generally we do not refer to Φ explicitly and the fact that v and w are end points of e is denoted as $e = (v, w)$.

A graph will be denoted as $G = (V, E)$ where the incidence mapping remains implicit. If V and E are both finite, G is called finite. For our descriptive purposes we are concerned only with finite graphs. A graph is said to be simple if at most one edge joins any pair of vertices and each edge is incident to two distinct vertices.

If the vertices of a graph represent components of individual

activities, and the edges are assigned an orientation, then in this case the incidences relations within the graph reflect the way in which execution of certain activities is contingent upon completion of other tasks. When an orientation is assigned to edges we have a directed graph.

A finite sequence $e_1, e_2, \ldots, e_n$ of (not necessarily distinct) edges of a graph is an edge progression of length n if there exists an appropriate sequence of n+1 (not necessarily distinct) vertices $v_0, v_1, v_2, \ldots, v_n$ such that $e_i = (v_{i-1}, v_i)$ for $i = 1, 2, \ldots, n$. The edge progression is said to be closed if $v_0 = v_n$ and open if $v_0 \neq v_n$.

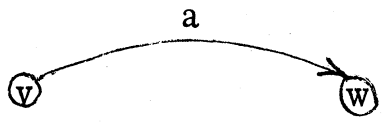
When the elements of a progression represent distinct edges, the edge progression is called a chain progression if it is open and a circuit progression if it is closed. The set of edges itself, without regard to sequencing is said to constitute a chain in the former case and a circuit in the latter.

With simple graphs, if all n+1 vertices $v_0, v_1, v_2, \ldots, v_n$ are distinct, the set of edges is called a simple chain. If $v_0 = v_n$ but the vertices otherwise distinct the unordered set of edges is said to constitute a simple circuit.

A graph is said to be connected if every pair of distinct vertices are joined by atleast one chain. A graph is said to be a tree if it is connected and has no circuits.

When dealing with directed graphs we replace the set E with a

set of "oriented edges" A called arcs, and B = (V, A). Incidence for any $a \in A$ is again designated as $a = (v, w)$, where a is said to have v as its initial vertex and w as its terminal vertex.



An arc progression of length n is a sequence of (not necessarily distinct) arcs $a_1, a_2, \ldots, a_n$ such that for an appropriate sequence of n+1 vertices $v_0, v_1, \ldots, v_n$ we have $a_i = (v_{i-1}, v_i)$ for $i = 1, 2, \ldots n$. The arc progression is closed if $v_0 = v_n$ and open if $v_0 \neq v_n$. An arc progression in which no arc is repeated is called a path progression or cycle progression, depending on whether it is open or closed. The corresponding set of arcs, is called a path or cycle respectively. A directed graph is said to be cyclic if it contains at least one cycle, and acyclic otherwise.

When the term "tree" is applied without qualification to a directed graph it is understood that arc directions are ignored and the associated undirected graph is being described. Some writers tend to use the terminology node and arc when discussing directed graphs and vertex and edge when dealing with undirected graphs. We tend to use vertex and edge in both contexts and will specify when necessary if the graph is oriented.

We will frequently refer to the connection matrix C of a graph.

It is defined as a matrix $C = \{c_{ij}\}$ where

$$c_{ij} = 1, \quad (v_i, v_j) \in E;$$
$$0, \quad \text{otherwise}$$

An analogous matrix can be defined for directed graphs, it is the incidence matrix [6] $M = \{m_{ij}\}$ which is defined as,

$$m_{ij} = 1 \quad \text{if } (v_i, v_j) \in A,$$
$$= -1 \quad \text{if } (v_j, v_i) \in A,$$
$$= 0 \quad \text{otherwise.}$$

BIBLIOGRAPHY

1. Arden, B. W., Galler, B. A., O'Brien, T. C. and Westervelt, F. H., "Program and Addressing Structure in a Time-Sharing Environment", Journal of the ACM, 13 (January, 1966), 1-16.

2. Asher, D. T., "A Linear Programming Model for the Allocation of R and D Efforts", IRE Transactions on Engineering Management, EM-9 (December, 1962), 154-157.

3. Balas, E., "An Additive Algorithm for Solving Linear Programs with Zero-One Variables", Operations Research, 13 (July-August, 1965), 517-546.

4. Balinski, M., "On a Selection Problem", Management Science 17, 3 (November, 1970), 230-231.

5. Beged Dov, A. G., "Optimal Assignment of Research and Development Projects in a Large Company Using an Integer Programming Model", IEEE Transactions in Engineering Management, EM-12, 4 (December, 1965), 138-142.

6. Berge, C., The Theory of Graphs and Its Applications, John Wiley and Sons, 1962.

7. Busacker, R., and Saaty, T., Finite Graphs and Networks An Introduction with Applications, McGraw-Hill, 1965.

8. Cabot, A. V., "An Enumeration Algorithm for Knopsack Problems", Operations Research, v8, 2 (March-April, 1970), 306-311.

9. Dantzig, G., Linear Programming and Extensions, Princeton University Press, 1963.

10. Denning, P. J., "The Working Set Model for Program Behavior", Communications of the ACM, 11 (may, 1968), 323-333.

11. Dennis, J. B., "Segmentation and the Design of Multiprogrammed Computer Systems", Journal of the ACM, 12(October, 1965), 589-602.

12. Edwards, A.W. F., and Cavalli-Sforza, L. L., "A Method for Cluster Analysis," Biometrics, (June 1965) 362-375.

13. Estrin, G. and Martin, D. "Models of Computations and Systems-Evaluation of Vertex Probabilities in Graph Models of Computations", Journal of the ACM, 14 (April, 1967), 281-299.

14. Everett, H., "Generalized Largrange Multipliers Methods for Solving Problems of Optimum Allocation of Resources", Operations Research, 11, (May-June, 1963), 399-417.

15. Geoffrion, A., "An Improved Implicit Enumeration Approach for Integer Programming", Memo RM-5644-PR, Rand Corp., Santa Monica, California, (June, 1968).

16. Gomory, R. and Hu, T., "Multi-Terminal Network Flows", Journal of the SIAM 9, 4 (December, 1961), 551-570.

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

18. Gue, R., and Liggett, J., "Analysis of Algorithms for the Zero-One Programming Problem", Communications of the ACM, 11, 12 (December, 1968), 837-849.

19. Hammer, P. and Rudeanu, S., Boolean Methods in Operations Research and Related Areas, Springer-Verlag, 1968.

20. Informatics Incorporated, "Program Paging and Operating Algorithms", Technical Documentary Report, TR 68-793-1 (September, 1968).

21. Jensen, R., "A Dynamic Programming Algorithm for Cluster Analysis", Operations Research, 17 (November-December, 1969), 1033-1057.

22. Karp, R. M., "A Note on the Application of Graph Theory to Digital Computer Programming", Information and Control, 3 (September, 1960), 179-190.

23. Kemeney, J. G. and Snell, J. L., Finite Markov Chains, Princeton, New Jersey, D. Van Nostrand Co., 1960.

24. Kernighan, B., "Some Graph Partitioning Problems Related to Program Segmentation", Ph. D. Thesis, Princeton, (January, 1969).

25. Kernighan, B. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs", The Bell Systems Technical Journal, 49, 2 (February, 1970), 291-307.

26. Kral, J., "The Formulation of the Problem of Program Segmentation in Terms of Pseudoboolean Programming", Kybernetika Cislo 1, Rocnik (January, 1968), 6-11.

27. Kral, J., "One Way of Estimating Frequencies of Jumps in a Program", Communications of the ACM, 11 (July, 1968), 475-480.

28. Kral, J., "To the Problem of Segmentation of Programs", Proceedings Information Processing Machine Symposium, Prague, Czechoslovakia, (September, 1964).

29. Lawler, E., "On the Optimal Partitioning of a Family of Subsets of a Finite Set", to be published.

30. Lawler, E., Notes on Combinatorial Optimization.

31. Lawler, E. L. and Bell, M. D., "A Method for Solving Discrete Optimization Problems", Operations Research, 14 (November - December, 1966), 1098-1122.

32. Lawler, E., Levitt, K., and Turner, J., "Module Clustering to Minimize Delay in Digital Networks", IEEE Transactions on Computers, C-18 (January, 1969) 47-57.

33. Lawler, E. and Wood, D., "Branch and Bound Methods: A Survey", Operations Research, 14, 4 (July - August, 1966), 699-719.

34. Marimont, R. B., "Applications of Graphs and Boolean Matrices to Computer Programming", SIAM Rev., 2 (October, 1960), 259-268.

35. Nemhauser, G. and Ullmann, Z., "A Note on the Generalized Lagrange Multiplier Solution to an Integer Programming Problem", Operations Research, 16 (March - April, 1968), 450-452.

36. Pankhurst, R. J., "Program Overlay Techniques", Communications of the ACM, 11 (February, 1968), 119-125.

37. Pinkerton, T., "The MTS Data Collection Facility", Memorandum 18, Concomp Project, University of Michigan, (June, 1968).

38. Pinkerton, T. B., "Program Behavior and Control in Virtual Storage Computer Systems", Technical Report No. 4, Concomp Project, University of Michigan, 1968.

39. Ramamoorthy, C. V., "The Analytic Design of a Dynamic Look-Ahead and Program Segmenting System for Multiprogrammed Computers", Proc. ACM 21st Natl. Conf., (August, 1966), 229-239.

40. Ramamoorthy, C. V., "Discrete Markov Analysis of Computer Programs", Proc. ACM 20th Natl. Conf., (August, 1965), 386-392.

41. Randell, B. and Kuehner, C. J., "Dynamic Storage Allocation Systems", Communications of the ACM, 11 (May, 1968), 297-306.

42. Rubin, J., "Optimal Classification into Groups: An Approach for Solving the Taxonomy Problems", Journal of Theoretical Biology, 15 (1967) 103-144.

43. Sebestyen, G., Decision-Making Processes in Pattern Recognition, Macmillian Company, New York, 1962.

44. Shepherd, M. J., and Willmott, A. J., "Cluster Analysis on the Atlas Computer", Computer Journal, 11 (1968), 57-62.

45. Smithies, A., "A Conceptual Framework for the Program Budget", Systems, Organizations, Analysis, Management, McGraw-Hill, 1969, 163-182.

46. Sokal, R., and Sneath, P., Principles of Numerical Taxonomy, Freeman, 1963.