IMPERIAL COLLEGE OF SCIENCE AND TECHNOLOGY

(University of London)

Department of Management Science

SOME GRAPH THEORY ALGORITHMS FOR OPERATIONAL RESEARCH

by

Peter Brooker

A thesis submitted in partial fulfilment of the requirements for the M.Phil degree and the DIC

C O N T E N T S

L I S T O F F I G U R E S

P R E F A C E

The material in this thesis has been organised as follows. Chapters are divided into numbered sections and equations are numbered consecutively through each chapter. In the text reference is made both to sections and particular equations. References to published work are collected at the end of each chapter.

Most of the terms used in the thesis are fully explained in the text. The exceptions to this rule are terms which require some detail to explain fully, e.g., Kuratowski graph, or are merely descriptive, e.g., Regular graph.

A good general book on Graph Theory is:

F. Harary, 'Graph Theory', Addison-Wesley (Mass.), 1971.

Explanations of terms used in Network Theory can be found in:

H. Frank and I.T. Frisch, 'Communication, Transmission and Transportation Networks', Addison-Wesley (Mass.), 1971.

A B S T R A C T

The thesis is concerned with computer algorithms for problems in Graph Theory. It is divided into four chapters.

Chapter I is a brief summary of the concepts of Graph Theory plus an examination of algorithms which will be used as building blocks for the more complicated methods to be described.

In Chapter II we consider the problem of partitioning a link-weighted graph in two parts, each of which is constrained in size by the (given) number of vertices that the part can contain. This is a special case of the general problem of partitioning a graph into k parts with size constraints and which appears in a number of very diverse problem areas.

We present a tree search method for solving the 'partition in two' problem, with the objective of minimising the sum of the costs of the cut links. The method employs a maximum-flow subalgorithm for establishing bounds during the search, and uses the concept of the longest spanning tree of a graph to direct the forward branching steps. The size of the search is reduced further by a test (based on the solution of a special 'knapsack' type problem) and which is designed to detect potential size-infeasibility early on in the search.

Chapter III is concerned with the optimal expansion of a given network. Given an existing network, a list of links which could be added to the network, the link costs and capacities and an available budget, the problem considered

is one of choosing which links to add to the network in order to maximise the maximum flow from a source s to a sink t, subject to the budgetary constraint. This problem appears in a large number of practical situations which arise in connection with the expansion of electricity or gas supply, telephone, road or rail networks. The method used is an efficient tree search algorithm using bounds calculated by a dynamic programming procedure which are very effective in limiting the solution space explicitly searched.

In Chapter IV, we examine the p-median problem - this is concerned with the location of facilities on a graph, so that facilities are placed in such a way as to minimise the sum of the shortest distances from the vertices of the graph to the nearest facility.

The problem arises in the location of switching centres, electric power network substations and other contexts. We survey the literature and describe a direct tree search algorithm with an effective 'penalty' bound.

We discuss computing results for the problems described above. In an Appendix, we list a computer program for the p-median problem, and discuss the general nature of computer programs in Graph Theory - with special reference to implicit enumeration techniques.

A C K N O W L E D G E M E N T S

It is a pleasure to thank Dr. Nicos Christofides for encouragement and guidance throughout this work, and for permission to include in Chapters II and III material which was published jointly with him.

I would also like to thank Robin Hewins for reading through the drafts of this thesis, during which he made several valuable suggestions which have made the final form possible.

I wish to thank all past and present members of the Department who have helped me with their remarks and suggestions in the past few years.

Last, but not least, I would like to thank Mrs. H. Arul-Pragasam and Miss C. Todorovitch for their excellent typing of this thesis.

C H A P T E R I

AN INTRODUCTION TO CONCEPTS OF GRAPH AND NETWORK THEORY

1.1 Some Definitions

The following chapters are concerned with some problems of graph and network theory. In this chapter we will give basic definitions for the framework of our discussions and sketch some simple algorithms we shall use in the following work.

A graph G is defined as a collection of links and vertices, so that $G = (X,A)$ for a set of vertices $x_i \varepsilon X$ and a set of links $e_{ij} = (x_i,x_j) \varepsilon A$ joining some, or all, of these vertices. We will make a distinction between 'graph theory' and 'network theory' as follows: a graph defines the purely structural relationships between the vertices, while a network bears also the quantitative characteristics of the vertices and links. Most of our interest is in the latter theory.

An arc is another name for a link, while vertices are also known as nodes or points. A link with its two end vertices is usually called a branch, in the present work we exclude the possibility of self-loops, i.e., branches with non-distinct vertices. A graph in which it is possible to reach any vertex from any other vertex along branches in the graph is said to be connected; otherwise it is unconnected. We define a chain between vertices x_i and x_j as being a connected sequence of branches that lead from x_i to x_j such that each vertex in the sequence is encountered only once. A link may have an orientation, and then it is known as a

directed link. A directed chain has all branch orientations leading from x_i to x_j. If some of the branches are traversed in the opposite direction the term path is used. A chain whose terminal vertices coincide is called a loop or cycle.

A subgraph H = (X',A') of a graph G = (X,A) is a graph whose $X' \subseteq X$ and $A' \subseteq A$. A spanning tree is a connected subgraph of all vertices which contains no loops; it is obvious that every connected graph must have at least one spanning tree. For a graph with n vertices the tree contains exactly n-1 links, although a graph may have several spanning trees. The links of G that are not in the tree are known as chords.

1.2 Maximum Flows and Minimum Cuts {1}

Let us associate a scalar c_{ij} (the 'capacity') with each link (x_i,x_j) of the graph G, with the c_{ij} all positive. The maximum flow problem is as follows. Maximise v subject to

$$\sum_i x_{ij} - \sum_k x_{jk} = \begin{cases} -v & \text{if } j=s \\ 0 & \text{if } j\neq s,t \\ v & \text{if } j=t \end{cases} \qquad (1)$$

$$0 \leq x_{ij} \leq c_{ij} \qquad \text{for all } i,j \qquad (2)$$

for a maximum flow from vertex x_s to vertex x_t. These two vertices are known as the source and the sink respectively. The network can be considered as a pipeline system with the links representing pipelines, the source being the inlet of the water, the sink being the outlet of the water, and all the other vertices just being junctions between pipelines. The capacity of each link can be supposed to indicate the cross-sectional area of the pipeline. With such a pipeline

system we are interested in the maximum flow that can be put through it from the source to the sink. The flow in link e_{ij} is x_{ij} which is bounded above by c_{ij} and is always non-negative. Our analogy with a water pipeline system is not complete as stated because we need valves at every vertex to prevent liquid seeping back at every vertex down incident links.

It is clear that the maximum flow value in any network can be found by solving a linear program with the objective function $v = \sum_j x_{sj}$ with the constraints (1) and (2). As this is a very special case of a linear program, however, more efficient algorithms than the simplex method can be used. The method we will describe is due to Ford and Fulkerson and is known as the 'Max-Flow Min-Cut' theorem.

If the network consists of a single chain going from source to sink, then the maximum amount of flow that can be put through the network is clearly limited by the link with the minimum capacity c_{ij} of all the links in the chain. Thus the link with the minimum capacity is the bottleneck of the network, this can be generalised for an arbitrary network using the concept of a network cut. A cut $K = (Y, \bar{Y})$, that is a partitioning of the network into two disjoint subsets of vertices Y and $\bar{Y}$. The capacity or value of the cut is the sum of the capacities of the links which have one vertex in Y and the other in $\bar{Y}$, in the case of a directed network only those capacities of links which lead from Y to $\bar{Y}$ are counted in this sum. Thus we have:-

$$\text{capacity of cut } K = \sum_{\substack{x_i \in Y \\ x_j \in \bar{Y}}} c_{ij} \qquad (3)$$

From constraints (1) and (2), it is clear that the maximum flow value v is less than or equal to the capacity of any cut separating x_s from x_t. In fact Ford and Fulkerson have proved that the maximum flow value is always equal to the minimum capacity of all of the cuts separating x_s and x_t. This is the central theorem of network flow theory. In the next section we present the labelling algorithm for constructing such a maximum flow.

1.3 The Labelling Algorithm for Maximum Flow

In this section vertex x_i if referred to by the index i for convenience.

Labelling Routine:

Step 1: Label s by (s, +, w(s) = ∞). s is now labelled and unscanned and all other vertices are unlabelled and unscanned.

Step 2: Select any labelled and unscanned vertex, i.

a) For all j such that $e_{ji} \epsilon A$, and $x_{ji} > 0$, where x_{ji} is the flow in line (j, i), label j by (i, -, w(j)) where w(j) is the minimum of w(i) and x_{ji}. Then j is labelled and unscanned.

b) For all j such that $e_{ij} \epsilon A$, j is unlabelled, and $c_{ij} > x_{ij}$ label j by (i, +, w(j)) where w(j) is the minimum of w(i) and $(c_{ij} - x_{ij})$. Then j is labelled and unscanned.

Change the label on i by circling the + or - entry. Then i is now labelled and scanned.

Step 3: Repeat step 2 until t is labelled or until

no more labels can be assigned. In the latter case the algorithm terminates. In the former case proceed to the augmentation routine.

Augmentation Routine:

Step 1: Let z = t and go to step 2 of the augmentation routine.

Step 2: If the label on z is (q, +, w), increase x_{qz} by w(t). If the label on z is (q, -, w), decrease x_{zq} by w(t).

Step 3: If q = s, erase all labels and return to step 1 of the labelling routine. Otherwise, let z = q and return to step 2 of the augmentation routine.

The labelling algorithm as described above converges to the correct limit if the link capacities are integers. If some or all of the c_{ij} are fractional, the algorithm may not be a finite one or may converge to a wrong limit. All the work in this thesis assumes integer capacities, although there are modifications which can deal with irrational numbers.

For the sake of clarity we have described the algorithm over the diagram of the network. In the following chapters the labelling and flow augmentation cycles are carried out over the matrix representation of a network in a digital computer.

1.4 The Shortest Spanning Tree of an Undirected Graph

In Chapter II we use the calculation of the longest spanning tree of a graph in our algorithm. There are several

methods for determining such a tree, that is, one whose sum of link weights is a maximum. These algorithms are usually framed in terms of the complementary problem - that of the shortest spanning tree (SST). All that is required to find the longest spanning tree is to change the sign of the link weights of the graph.

The algorithm we use is due to Prim (2). It uses the concept of a subtree, which is a tree defined on a subset of the vertices of the graph, with the associated links. This algorithm produces an SST by growing only one subtree T_s (say) containing more than a single vertex and considering the remaining single vertices to form one subtree each. Subtree T_s is then grown continuously by adjoining that link (x_i, x_j), $x_i \epsilon T_s$, $x_j \notin T_s$ with the minimum cost c_{ij} until $(n - 1)$ links are added and T_s becomes the required SST.

The algorithm proceeds by labelling each vertex $x_j \notin T_s$ with (α_j, β_j) where, at any step, α_j is the vertex of T_s nearest to vertex x_j and β_j is the length of this link (α_j, x_j). At any one step during the algorithm that vertex - say x_{j*} - with the smallest β_j is appended to T_s by the addition of link (α_{j*}, x_{j*}). Since T_s has now acquired a new vertex x_{j*}, the labels (α_j, β_j) for those vertices $x_j \notin T_s$ may now need updating (if, for example, $c(x_j, x_{j*})$ is less than the existing label β_j), and the process continued. This labelling procedure can be seen to be very similar to that used for the shortest path problem using the Dijkstra algorithm (3).

The algorithm is as follows:-

Step 1: Let $T_s = (x_s)$, where x_s is any arbitrarily chosen vertex, and $A_s = \phi$. (A_s will be the

set of links forming the SST).

Step 2: For all $x_j \notin T_s$ find a vertex $\alpha_j \in T_s$ so that:

$$c(\alpha_j, x_j) = \min_{x_i \in T_s} (c(x_i, x_j)) = \beta_j$$

and set the label of x_j as (α_j, β_j).

If no such vertex α_j can be found, i.e., if $\Gamma(x_j) \cap T_s = \phi$,† set the label of x_j as $(0, \infty)$.

Step 3: Choose that vertex x_{j*} so that

$$\beta_{j*} = \min_{x_j \notin T_s} (\beta_j)$$

Update $T_s = T_s \cup (x_{j*})$, $A_s = A_s \cup ((\alpha_{j*}, x_{j*}))$.

If $|T_s| = n$, stop. The links in A_s form the SST.

If $|T_s| \neq n$, go to step 4.

Step 4: For all $x_j \notin T_s$ and $x_j \in \Gamma(x_{j*})$ update labels as follows:

If $\beta_j > c(x_{j*}, x_j)$, set $\beta_j = c(x_{j*}, x_j)$, $\alpha_j = x_{j*}$ and return to step 3.

If $\beta_j \leq c(x_{j*}, x_j)$ go to step 3.

1.5 Branch and Bound Methods

The algorithms described in the following chapters are all Branch and Bound algorithms. Branch and Bound methods use both implicit and explicit enumeration, as distinct from methods such a Dynamic Programming which basically is an efficient way of performing explicit enumeration.

In general the method involves searching the space

---

† Here, $\Gamma(x_i)$ is the set of vertices which are directly connected to vertex x_i by a link. Thus if a graph contains a link (x_i, x_j), x_j will be a member of the set $\Gamma(x_i)$ and vice versa.

of feasible solutions by partitioning into smaller and smaller subsets - a lower bound (in the case of minimization) being calculated for the cost of the solutions within each subset. Following each partition the subsets with bounds that exceed the cost of a known feasible solution are excluded from all the succeeding partitions. The process is continued until a feasible solution is found such that its cost is no greater than the bound for any subset.

To illustrate the method we will describe the algorithm for the Travelling Salesman problem due to Little et al. (4). The problem arises as follows; consider a travelling salesman who has to visit n cities or customers. Initially he is at city 1 (say) and must go to all the other cities once only and then return to his base city 1. The 'cost' of travelling from city i to city j is given as d_{ij}, this could be distance, time, money, etc.; the problem is to minimize the total distance of the journeying.

For n cities there are (n - 1)! possible tours for a general cost matrix, and half this number for a symmetric one (each trip may be reversed in direction). From Stirling's formula we know that n! behaves as $\sqrt{2\pi n}(n/e)^n$ for large values of n, so that enumerating all tours to find the optimal solution becomes increasingly less satisfactory in terms of computer time and storage capacity.

What we have called a tour above is more technically known as a Hamiltonian circuit (cycle) - a chain which passes through all the vertices of the system and which is elementary in that no vertex occurs twice on the chain apart from the first and last vertices.

There are two useful theorems which are basic to

the construction of the Branch and Bound algorithm - the Reduction Theorem and the Penalty Theorem.

Reduction Theorem:

If we subtract a constant e_i from all entries in row i, and a constant f_j from all entries in column j of the 'cost' matrix (d_{ij}) then the optimal tour under the final matrix

$$d'_{ij} = d_{ij} - e_i - f_j$$

is the same as that for the original matrix c, while the total cost Z'(t) is given by

$$Z'(t) = Z(t) - \sum_i e_i - \sum_j f_j$$

where t represents a tour and Z(t) is the sum of the link costs on the tour. The theorem is obvious - as every city has to be 'gone to' and 'come from' in the tour. The numbers e_i and f_j are known as 'reducing constants'.

Penalty Theorem:

If a certain tour does not contain the link (x_a, x_b), then the cost of that tour is certainly not less than the sum of the minimum elements in row a and column b of the cost matrix. Again the theorem is obvious when we consider the necessary alternative costs. We denote the sum of minimum elements by p_{ab} - it is known as the 'penalty cost'.

The method of Little et al. uses both of these theorems. For a given cost matrix the set of tours is split up into subsets, the subset size being decreased at each stage and a lower bound being calculated for the tour cost at each split. The bounds guide the partitioning of the

subsets so that eventually we arrive at a tour whose cost is less than the lower bounds. The subsets of tours can be represented as the nodes of a tree, which state whether or not a certain journey is included in that subset, and the branches of the tree indicate the partitioning.

Operations at each branching are identical. To highlight those links between cities which we would particularly like to choose (for instance those with low cost) the cost matrix is reduced by row and column until there is a zero in each row and column, with a total reduction cost of h. It is clear that the zeroes correspond to the cheapest links and it is likely that at least some of them may be in the optimal solution.

Initially, h is a lower bound for the tour cost, while in later stages it is added on to the lower bound $Z_b(t)$.

We now split the tours into two subsets, one of which includes the link (x_i, x_j), the other labelled $(\overline{x_i, x_j})$ not containing it. There are many ways of picking (x_i, x_j) - Little et al. choose it to be the link with greatest penalty cost. Thus for branching from vertex X to vertex pairs $Y((x_i, x_j))$ and $\overline{Y}((\overline{x_i, x_j}))$ we know that a lower bound on the $\overline{Y}$ vertex is $Z_b(\overline{Y}) = Z_b(t) + p_{ij}$. Initially, however, we do not go to $\overline{Y}$, but follow positive vertices such as Y, until a tour is completed so that we have a tour cost for comparison purposes.

Consider the vertex Y in the tree search, we must cancel out row i and column j to prevent two journeys from x_i or to x_j. To avoid a subtour the element c_{ji} must be put equal to infinity, similarly if (x_i, x_j) joins onto a chain

the link between the two end cities must be excluded. The process of splitting begins again, and continues until the cost matrix is 2 x 2 when the link to be chosen is obvious.

It is possible that the solution is optimal. If it is not we must go back up the tree and branch from vertices with bounds less than Z_0 (the cost of the feasible solution just found). There are three main ways of searching the tree:-

(1) Branch from first vertex in the tree with bound less than Z_0.

(2) Branch from vertex with lowest bound.

(3) Branch from nearest vertex to the best solution so far with bound less than Z_0.

Note that the cost matrix has to be reset at this branching, all the operations on the positive side (for want of a better phrase) of the tree being 'undone'. Little et al. adopt the second method while the third choice is the only one satisfactory for a large problem as the tree (or at least the branches between the 'free' and terminal positive vertices) can be thrown away. Thus there is always enough computer storage space as the same core space can be used over and over again.

There are several variations of this method, for instance lower bounds can be found by considering the shortest spanning tree of the city network - details of these and computational aspects are given by Eilon et al. (5).

REFERENCES

1. L.R. FORD and D.R. FULKERSON. "Maximal flow through a network". Can. J. Math., 8(3), (1956), pp. 399-404.

2. R.C. PRIM. "Shortest connection networks and some generalizations". Bell Syst. Tech. J1, 36, (1957), pp. 1389-1401.

3. E.W. DIJKSTRA. "A note on two problems in connection with graphs". Numerische Mathematik, 1, (1959), pp. 269-271.

4. J.D.C. LITTLE, K.G. MURTY, D.W. SWEENEY and C. KAREL. "An algorithm for the travelling salesman problem". Ops. Res., 11, (1963), pp. 972-989.

5. S. EILON, C.D.T. WATSON-GANDY and N. CHRISTOFIDES. "Distribution management: mathematical modelling and practical analysis". (1971), Griffin, London.

C H A P T E R I I

THE OPTIMAL PARTITIONING OF A GRAPH

2.1 Introduction

A graph G may, for example, represent an electronic circuit, with the graph vertices corresponding to circuit components and the links corresponding to wires (or other connections) between these components. In the physical implementation of the electronic circuit, one may - if the circuit contains many components - require their placement on two or more circuit boards because of the limitations on the number of components per board. An important problem is then to allocate components to boards so as to minimise the total cost of the links (wires) between boards. Very often connections between boards must be made externally (and often manually) whereas connections of components within a board are 'printed' onto the board itself. Hence, if the cost of every link of the graph corresponding to the electronic circuit is taken as unity, the placement corresponding to the minimum number of interboard wires represents a least cost partition of the graph. This minimises a major part of the production cost of the electronic circuit.

The problem of finding the least cost partition of a general graph into a number of subgraphs of restricted size appears in many other diverse fields. For example, in clustering analysis one usually tries to find clusters of objects (vertices in our terminology) where the total sum of the intervertex costs within the clusters

is minimised. This is obviously a maximisation version of our present problem. In the general area of numerical taxonomy as applied to biology the "clusters" represent groupings of objects and the additional condition often exists that any two objects in a cluster must be related, either directly or indirectly through a third object also in the cluster. In terms of the graph structure this implies that the partition must be such that each of the parts forms a connected graph.

A similar connectivity condition of the sub-graphs formed by the partition appears in the general problem of "districting". In this problem, an area is composed of a large number of elementary regions, and what is required is to form clusters of regions each cluster representing a subdivision (district) of the original area. Such problems appear in dividing an area amongst salesmen, political districting (17), vehicle scheduling (18) etc. The problem can be represented as a graph with the elementary regions as vertices and with links between vertices corresponding to neighbouring regions. Link "costs" are measures of similarity between regions, the nature of the measure depending on the type of problem. In these problems, what is always required is to have districts of limited size which are <u>contiguous</u> i.e. to partition the corresponding graph into subgraphs each of which is of limited size and is connected.

The method described in this chapter deals with the problem of partitioning a graph into two subgraphs with size constraints and with the connectivity constraints mentioned above.

The basic problem of graph partitioning with size constraints has been investigated by Kernighan and Lin (5), who gave an heuristic method of solution to the problem. Their procedure is based on the following fact. Given an arbitrary partition into subgraphs A and B, where $A = \{a_i\}$ and $B = \{b_i\}$, the a_i and b_i being general nodes of the graph G, there exists $A' \subseteq A$ and $B' \subseteq B$ such that interchanging the nodes A' and B' will create an optimal partition. Their algorithm thus starts with an arbitrary partition A, B and attempts to improve upon it.

The first step is to find elements $a_1' \in A$ and $b_1' \in B$ such that interchanging a_1' and b_1' will decrease the total link cost Z (see section 2.2) by an amount g_1. This interchange is then tentatively made. The procedure is repeated, where now any node which has been interchanged is not considered again for further interchange. For a graph with 2m nodes, this process is repeated m times from which a_1', a_2', .., a_m' and b_1', b_2', .., b_m' and g_1, g_2, .., g_m are determined. Note that the g_i may be positive, zero, or negative - we chose the g_i so that they are monotonically decreasing as i increases. It is obvious that

$$\sum_{i=1}^{m} g_i = 0$$

since after the last interchange A and B have swopped over.

Let k be the index for which

$$\sum_{i=1}^{k} g_i = Z$$

is maximum. Let

$$X = \{a_1', a_2', .., a_k'\} \text{ and } Y = \{b_1', b_2', .., b_k'\}.$$

Starting with the original A and B, interchanging the sets of nodes $X \subseteq A$ and $Y \subseteq B$ results in a decrease in Z of

value G. If G is greater than zero, the interchange is carried out, thereby producing a new A and B. If Z is zero, the procedure terminates with a local optimal partition. The authors also discuss the extension of this procedure to the problem with A and B of different sizes.

Computing times are said to increase as $m^2 \log m$. Note that this method does not guarantee a global optimum, though the answers are said to be close in Z value to one.

An optimal method of partitioning for graphs with unit link costs was given by Gorinshteyn (6), under the name of 'ordered sorting', which is essentially a branch and bound algorithm. Although no computational results are mentioned in this reference, it is highly unlikely that the method would be suitable for graphs that are not trivially small. Another optimal partitioning method for a restricted class of partitions was given by Kernighan (7). His method deals with graphs where the vertices are in a sequence and partitioning is meant to be the splitting into subsequences without affecting the relative position of the vertices. Such problems appear in partitioning computer programs to be assigned onto pages for operation in a paging machine. The method is fast, but does not generalise to our problem.

The algorithm presented here is a tree search procedure which produces an optimal general partitioning of a link-weighted graph. The efficiency of such an algorithm (like any other tree search method) is based on the effectiveness of bounds and infeasibility tests in limiting the size of the search tree, and also on the choice of the tree branching strategy. Moreover, the

effectiveness of the calculated bounds or infeasibility tests may itself depend on the nature of the search. In the algorithm, the branching principle is chosen so as to increase the efficiency of the bound, which is derived from a maximum flow/minimum cut calculation. Infeasibility tests, designed to indicate early on that no size-feasible partition exists (given the decisions already made) are also used during the search. The branching strategy is chosen so as to produce "backtracking" as early as possible in the same spirit as the branching strategies used by Little et al for the travelling salesman problem (15).

Test results are given in the final section of this chapter for approximately 300 graphs from which it can be seen that the algorithm is quite efficient for medium sized (up to about 40 vertices), sparse graphs, but that as the density is increased, the efficiency is adversely affected. However, the graphs encountered in the majority of practical cases are sparse, and for these cases the proposed algorithm offers a means of solving (optimally) partitioning problems on graphs of up to 40 or so vertices.

2.2 The Problem

Consider the graph $G = (X, A)$ with the link cost matrix (c_{ij}). It is required to partition the set of vertices X into two sets:- Y and $\bar{Y} = X - Y$, so that the cost z, where

$$z = \sum_{\substack{x_i \in Y \\ x_j \in \bar{Y}}} c_{ij} \qquad (1)$$

is a minimum, and where

$$L \leq |Y| \leq U, \quad n - U \leq |\bar{Y}| \leq n - L \qquad (2)$$

We use $|S|$ to mean the number of elements - the cardinality - of a set S, L and U to be the lower and upper limit sizes of subset Y, and $n \equiv |X|$ to be the number of vertices of the original graph G. In addition, it is required that the two subgraphs $G_Y = (Y, A_Y)$, where $A_Y = \{e_{ij} | x_i, x_j \varepsilon Y\}$, and $G_{\bar{Y}} = (\bar{Y}, A_{\bar{Y}})$, where $A_{\bar{Y}} = \{e_{ij} | x_i, x_j \varepsilon \bar{Y}\}$, which are defined by the sets Y and $\bar{Y}$ respectively, should both be connected In other words, there should be a chain of links from every vertex in G_Y to every other vertex in G_Y, and similarly for $G_{\bar{Y}}$. This requirement ensures that each subgraph represents, in reality, a single entity and not merely a collection of disjoint parts. The connectivity and size constraints on G_Y and $G_{\bar{Y}}$ may not always be compatible and it is possible that no feasible partition of G satisfying both these constraints exists.

The partitioning problem defined by equations (1), (2) and the requirement that G_Y and $G_{\bar{Y}}$ be connected, can be formulated as a 0-1 programming problem. However, the problem cannot be solved efficiently in this way for any reasonably sized graphs, given the current state of 0-1 programming algorithms.

2.3 The Algorithm

The algorithm described in this section is essentially a tree search method with bounds on the value of the optimal solution (z* say) derived from maximum flow calculations. The relationship between the present problem and the maximum flow problem is quite obvious.

Thus, suppose that the present problem contained no size restrictions on Y and $\bar{Y}$, and that maximum flows Φ_{ij} are computed between every pair of vertices x_i and x_j - taking the costs c_{ij} as the link 'capacities'. This could be done efficiently by the method of Gomory and Hu (8). From the maximum-flow/minimum-cut theorem (9), we know that the cut of least cost (capacity) separating a vertex x_i from a vertex x_j, i.e. partitioning G in two, is equal to the maximum possible flow in G from x_i to x_j. Hence $\Phi^*_{ij} = \min_{i,j} (\Phi_{ij})$ would then be the value of z* and the cut corresponding to Φ^*_{ij} would indicate the minimum partition of G.

However, when the sizes of Y and $\bar{Y}$ are restricted, what is required is a different cut that has minimum cost and satisfies these conditions. It is well known (10) that this simple restriction alone makes the problem a very difficult one.

In the present method, we will build up the required minimum cut during a tree search. Now, at some stage of the search the decisions made so far will be stored in a vector S, which will sometimes be treated as an ordered list of signed elements (these elements will be links) and sometimes as an unordered set, the meaning being obvious from the usage. The search itself is a 'depth-first' procedure similar to that described by Geoffrion (11). Thus, if the link $e_{ij} = (x_i, x_j) \varepsilon S$, this is interpreted to mean that link e_{ij} has been chosen to be in the cut, or equivalently, that vertices x_i and x_j are one in Y and the other in $\bar{Y}$. Let $-e_{ij} \varepsilon S$ be taken to mean that e_{ij} is specifically excluded from the cut, i.e. that

x_i and x_j are both in Y or both in $\bar{Y}$. If $\pm e_{ij} \notin S$ no decision about the link has as yet been made. Thus, given an initial graph G = (X, A), a given set of decisions as represented by the set S then defines the subgraph G_S = (X, A_S), where

$$A_S = \{e_{ij} | e_{ij} \in A \text{ and } \notin S\}$$

and the link costs: $c_{ij}^S = c_{ij}$ if $e_{ij} \in A_S$ and $-e_{ij} \notin S$

otherwise $c_{ij}^S = \infty$ if $e_{ij} \in A_S$ and $-e_{ij} \in S$ (3)

The costs of the links in the cut so far will be denoted by

$$z_S = \sum_{e_{ij} \in S} c_{ij}$$

and z* will be the cost of the best feasible cut to date.

The basic features of the algorithm will now be described and some of the techniques used to execute the various steps are given in greater detail in later sections followed by a description of the algorithm.

2.3.1 The basic method

The algorithm proceeds by adding links into S as positive entries, in an attempt to form a feasible cut. After each addition into S, a lower bound B on the cost of the optimal solution (given the decisions already made) is calculated from a number of maximum flow-minimum cut computations on the graph G_S defined in (3) above. If this bound B $\geq$ z*, the algorithm backtracks. If not, it proceeds to test for possible size-infeasibility that may have become inevitable given the decisions already made. If the infeasibility test also fails, then a new link is

chosen to augment S and the algorithm continues by calculating a new lower bound and so on.

What distinguishes the current algorithm are the bound calculations, the size infeasibility tests, which are effective and involve the solution of "knapsack problems" using a dynamic programming sub-algorithm, and the augmentation which is iterative and uses a longest spanning tree calculation to choose that link which is most likely to lead to an early backtracking.

2.3.1.1 Calculation of the bound Let $e_{ij} = (x_i, x_j)$ be a link entered positively in S. This implies that in any solution derived from S, we must have $x_i \epsilon Y$ and $x_j \epsilon \bar{Y}$ or vice versa. Hence by treating the link costs c_{ij} of G_S as capacities - see equation (3) - the maximum flow Φ_{ij} from x_i to x_j in G_S would (by the maximum flow/minimum cut theorem) be a lower bound on the extra cost required for partitioning G_S. Another value for the bound can be derived by performing a maximum flow calculation between the end vertices of another link of S. Thus a valid lower bound B is given by:

$$B = \max \{\Phi_{ij} | e_{ij} \epsilon S\} \qquad (4)$$

However, one need not perform as many flow calculations as there are positive entries in S in order to obtain the maximum in equation (4). If Q_{ij} is the cut corresponding to Φ_{ij} and if a link $e_{\alpha\beta} = (x_\alpha, x_\beta) \epsilon S$ spans Q_{ij} (i.e. x_α is in one subgraph and x_β in the other subgraph formed by the cutting of G_S by Q_{ij}), then the maximum flow $\Phi_{\alpha\beta}$ need not be calculated, since in terms of capacities $Q_{\alpha\beta} \leq Q_{ij}$ and hence $\Phi_{\alpha\beta} \leq \Phi_{ij}$. Thus, in

general, only a few flow calculations will be needed to establish a strong bound, since more and more possibilities are being excluded as the flow calculations continue.

Moreover, if at some stage (before the final value of $B_{''}$ as given by equation (4), is established) a value for B is reached so that $z_S + B \geq z^*$, one need not continue the calculation further and backtracking can occur. At an even more detailed level, since the final value Φ_{ij} of the maximum flow from x_i to x_j is derived by improving (monotonically) a feasible flow, further calculation of any Φ_{ij} is again unnecessary if at some stage $z_S + \Phi_{ij} \geq z^*$.

2.3.1.2 Infeasibility tests If the bound is successful in pruning the tree search, backtracking occurs i.e. the last link entered in S as a positive entry now has its sign reversed, i.e. it is specified as not being in the cut. Thus, at a general stage, some links in S will be negative entries and a graph $G'_S = (X, A'_S)$ can then be defined by these links so that:

$$A'_S = \{e_{ij} | e_{ij} \varepsilon A, -e_{ij} \varepsilon S\} \qquad (5)$$

G'_S consists of one or more connected components and all the vertices within a certain component must, by definition of G'_S, be wholly in Y or wholly in $\bar{Y}$. Thus, if the largest component, say G_o, has n_o vertices with $n_o > U$ then no feasible solution can result from any forward branchings in the tree search from S. Backtracking can then take place.

A stronger size-infeasibility test is possible at this stage. Let $n_1, n_2, \ldots, n_p$ be the number of

vertices of the p connected components forming G'_S. Then, although $n_o = \max \{n_r | r = 1, .., p\}$ may not be greater than U, the numbers n_r may still not be arrangeable into two size feasible sets Y and $\bar{Y}$. Suppose, for example, that G is a 20-vertex graph and that it is to be partitioned into two parts, each of which must contain between 9 and 11 vertices (inclusive). If at some stage the graph G'_S has four connected components containing 7, 5, 7 and 1 vertices, then it can be easily shown that no combination of these can lead to a size-infeasible partition.

Thus, let $\xi_r = 1$ if the vertices of component r lie in set Y and $\xi_r = 0$ otherwise. A 0-1 valued vector $\underline{\xi}$ must now exist (where $\underline{\xi} = (\xi_1, \xi_2, .., \xi_p)$) so that

$$L \leq \sum_{r=1}^{P} \xi_r n_r \leq U \qquad (6)$$

These two inequalities can be transformed into the following problem

$$\begin{aligned} &\text{Maximise} \quad \lambda = \sum_{r=1}^{P} \xi_r n_r, \\ &\text{subject to} \quad \sum_{r=1}^{P} \xi_r n_r \leq U \end{aligned} \qquad (7)$$

Problem (7) defines a special case of the well-known knapsack problem, where the coefficients of the objective function and the constraint are the same. If λ^*, the optimum value of λ, is less than L, then no feasible solution to constraints (6) exists and backtracking can occur. If $\lambda^* \geq L$, then infeasibility cannot be proved and the algorithm can continue. Since problem (7) is a very easy one to solve, (12), (13), this test is not computationally expensive as regards time or storage.

2.3.1.3 Augmentation When neither the bonding nor the infeasibility tests are successful in pruning the search, a forward branch will be necessary and some new link has to be chosen to be in the cut, i.e. entered into S. The choice of the next link to enter into S obviously affects the performance of the algorithm, and we want that link whose addition will cause a backtracking step to be taken as early as possible. This rule is analogous to the heuristic augmentation rule used by Little et al (15) in the case of the travelling salesman problem.

Considering the graph G_S defined by (3), let T be its maximum cost spanning tree.† Such a tree can be calculated very easily by one of many well-known algorithms (14). A property of every spanning tree of a graph is that there is only one path between any two vertices of the graph using only links in the spanning tree. Thus let T_{ij} be that subset of the links of T on the unique path from x_i to x_j.

(i) Since our final objective is to obtain an optimal partition of G into sets Y and $\bar{Y}$ and since a link $e_{ij} \epsilon S$ implies $x_i \epsilon Y$ and $x_j \epsilon \bar{Y}$ (or vice versa), it is quite apparent that at least one link $e_{rt} \epsilon T_{ij}$ must be in the final solution, otherwise a path from Y to $\bar{Y}$ would exist which is a contradiction. Hence,

$$\delta_{ij} = \min \{c_{rt} | e_{rt} \epsilon T_{ij} \text{ for a link } e_{ij} \epsilon S\} \qquad (8)$$

is the least additional cost required to separate x_i from x_j and

$$\delta = \max \{\delta_{ij} | e_{ij} \epsilon S\} \qquad (9)$$

is therefore the least additional cost required. Let i*,

---

† Recall that a spanning tree of a graph G with n vertices is composed of (n - 1) links of G, so that there is a path from every vertex to every other vertex of G using only the links in the tree.

j* be the values for i and j corresponding to the value of δ of equation (9) and let r* and t* be the values of r and t producing δ_{i*j*} in equation (8). The link e_{r*t*} is then chosen to augment S on the basis that its cost $c_{r*t*} = \delta$ produces the largest necessary increase to the objective function and the search can now be limited more effectively as we will see below.

It should be noted that equations (8) and (9) only apply when S is not equal to the null set Φ. If $S = \Phi$ then a similar augmenting rule can be used, so that the lnik e_{r*t*} is chosen where

$$c_{r*t*} = \min \{c_{rt} | e_{rt} \varepsilon T\} \qquad (10)$$

(ii) Now with e_{r*t*} entered into S, if there exists a link e_{ij} of G, with $\pm e_{ij} \varepsilon S$, such that $z_S + c_{ij} \geq z^*$, we can immediately eliminate $+e_{ij}$ from inclusion. We can then augment S by $-e_{ij}$ and mark it (to indicate that at a future backtracking step the alternative of entering $+e_{ij}$ into S need not be considered for the given S). This can be done for all other similar links. If any negative links are introduced into S in this way, the sizes of the connected components of the corresponding graph G'_S are checked again for infeasibility as was done in section 2 above.

(iii) Consider the graph G'_S and a link $e_{ij} \varepsilon A$, where vertex x_i is in connected component v and vertex x_j is in component $w (\neq v)$ of G'_S. If link e_{ij} is excluded from the solution cut, i.e. if $-e_{ij}$ is entered into S, and if $(n_v + n_w) > U$, then obviously a size infeasibility would occur and we can, therefore, augment S with $+e_{ij}$ and mark it. This can be done with all links e_{ij} for which $\pm e_{ij} \varepsilon S$ and for which $(n_v + n_w) > U$.

If for the new set S we have $z_S \geq z^*$, we can backtrack, otherwise continue as follows: the addition of the (positive) links in S would have increased z_S so that repetition of step (ii) above may now lead to further augmentation of S, which in turn may lead to further augmentation at (iii). This continues until no more links can be specified by iterating these procedures. Thus, it should be noted that the entry of a single link $e_{r^*t^*}$ into S at (i) above, can lead to many other forced augmentations of S. This is a very desirable feature indeed, since these forced augmentations are all marked, so that at the next backtracking step they are all eliminated, thus saving a great deal of tree searching.

2.3.2 Description of the algorithm

Initialisation: Step 0. Set $S \leftarrow \Phi$, $z^* \leftarrow \infty$.

Investigation: Step 1 (a). Form the graph G_S according to eq. (3).

Step 1 (b). If G_S is disconnected into two subgraphs G_Y and $G_{\bar{Y}}$, and $L \leq |Y| \leq U$ go to Step 2, otherwise go to Step 4.

Step 2. If $z_S \leq z^*$, go to Step 3, otherwise go to Step 7.

Step 3. Replace $z^* \leftarrow z_S$, store $S^* = \{e_{ij} | +e_{ij} \in S\}$ and go to Step 7.

Step 4. Calculate a lower bound B on the least possible cost of partitioning G_S, and let Q be the cut corresponding to B. If $z_S + B \geq z^*$, go to Step 7. If $z_S + B < z^*$ and the cut Q separates the set of

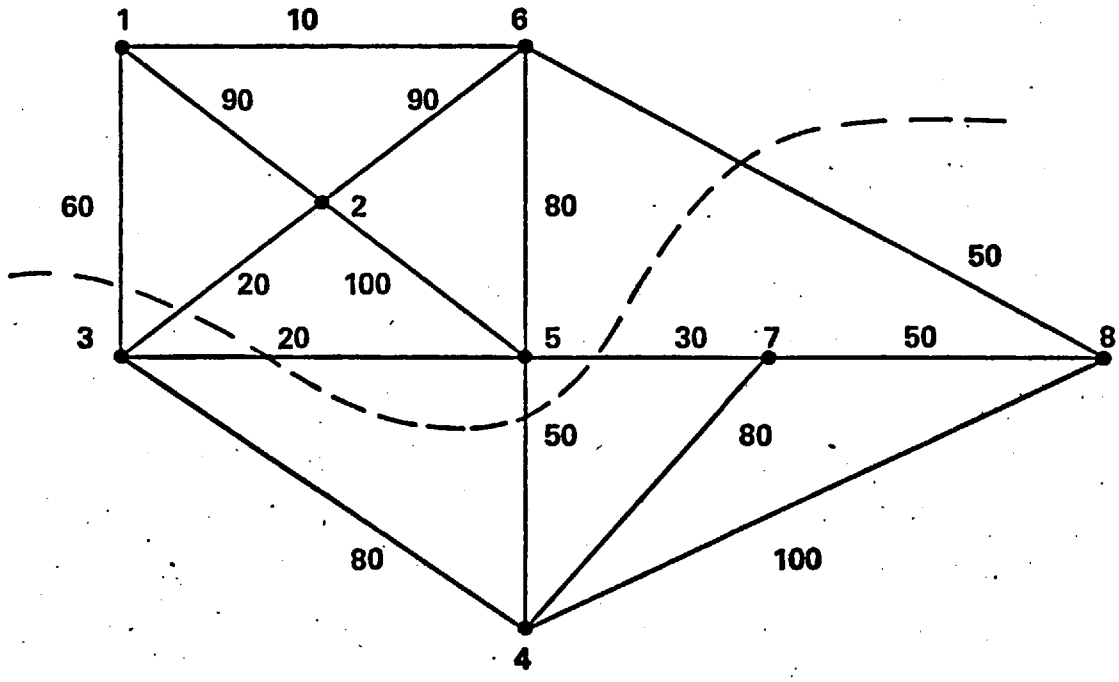

FIG 1. GRAPH FOR THE PARTITION EXAMPLE

vertices Y from the set $\bar{Y}$ with $L \leq |Y| \leq U$, set $z^* \leftarrow z_S + B$, $S^* = \{e_{ij} | +e_{ij} \in S\} \cup Q$, and go to Step 7. Otherwise go to Step 5.

Step 5. Form the graph $G'_S = (X, A'_S)$ according to equation (5) and apply the size feasibility test by solving problem (7). If infeasibility is proven, go to Step 7, otherwise go to Step 6.

Augmentation: Step 6. Augment S by a link $e_{r^*t^*}$ calculated according to equation (10). Iterate through stages (ii) and (iii) of Section 2.3.1.3 until no further forced augmentation of S is possible. Return to Step 1.

Backtracking: Step 7. Find the last element of S which is unmarked. If the element is of the form $+e_{ij}$, change it to $-e_{ij}$ and vice versa. Mark the element and remove from S all elements following it. Return to Step 1.

If no unmarked element exists in S, stop. The links in S* are the optimum partition of G, and the optimum value is z*.

2.4 Example

Consider the graph shown in Fig. 1, where the link costs are shown next to the links, and suppose it is required to find the minimal partition of the graph into two connected subgraphs of 4 vertices each.

The algorithm developed in the previous section will be followed and in addition the tree search will be

shown pictorially as the tree of Fig. 2 is evolved. The node numbers mentioned in what follows refer to the node numbering of Fig. 2.

In Fig. 2 the following information is given: within each circle, representing a tree node, the augmenting link is shown. Forced augmentations are shown in rectangles. At the top right hand of each node is the node number and at the bottom right hand is shown the calculated lower bound - whenever this is less than z^*. At each terminal node an explanation is given as to which step of the algorithm was responsible for the abandonment of any further search from that node.

Node 0. Initialisation. $S \leftarrow \Phi$, $z^* \leftarrow \infty$.

Node 1. The graph G_S is formed in Step 1(a). The first active step is the augmentation of Step 6. This is performed by the method outlined in section 2.3.1.3, i.e. the shortest link in the longest spanning tree is chosen to augment S. The longest spanning tree is shown in Fig. 3 and its shortest link is (1, 3).

Return to Step 1. The next active step is Step 4, where the flow calculation gives a size infeasible cut of cost 160 (= 100 + 60), by cutting vertex 1 from all the others.

Node 2. The next active step is 6, where S is augmented by link (4, 5), this being the shortest link in the longest spanning tree - updated following the exclusion of link (1, 3). Return to Step 1 and through to Step 4, where the flow calculation results in a feasible solution of cost $z^* = 230$,

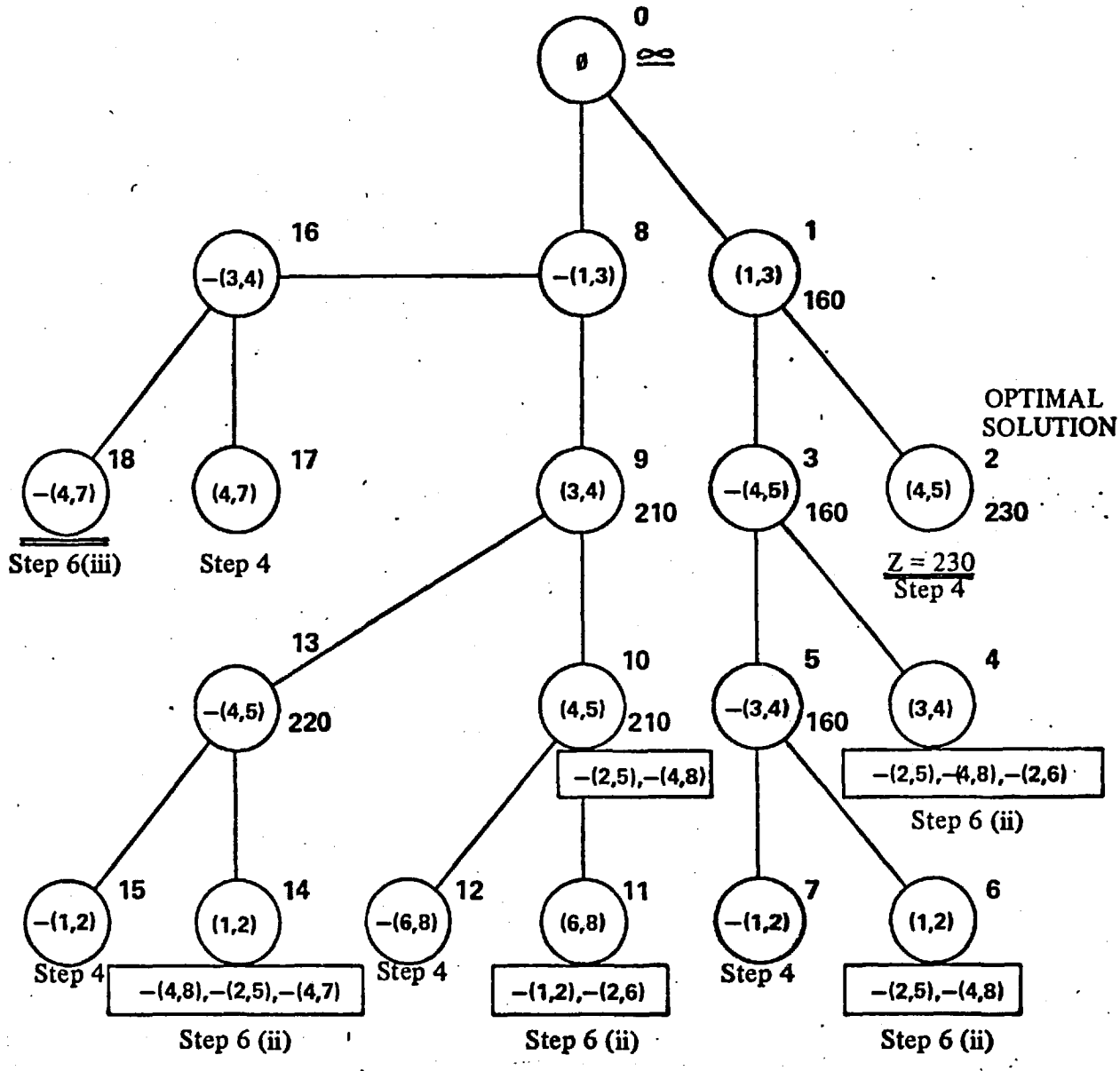

FIG. 2 : BRANCH AND BOUND TREE

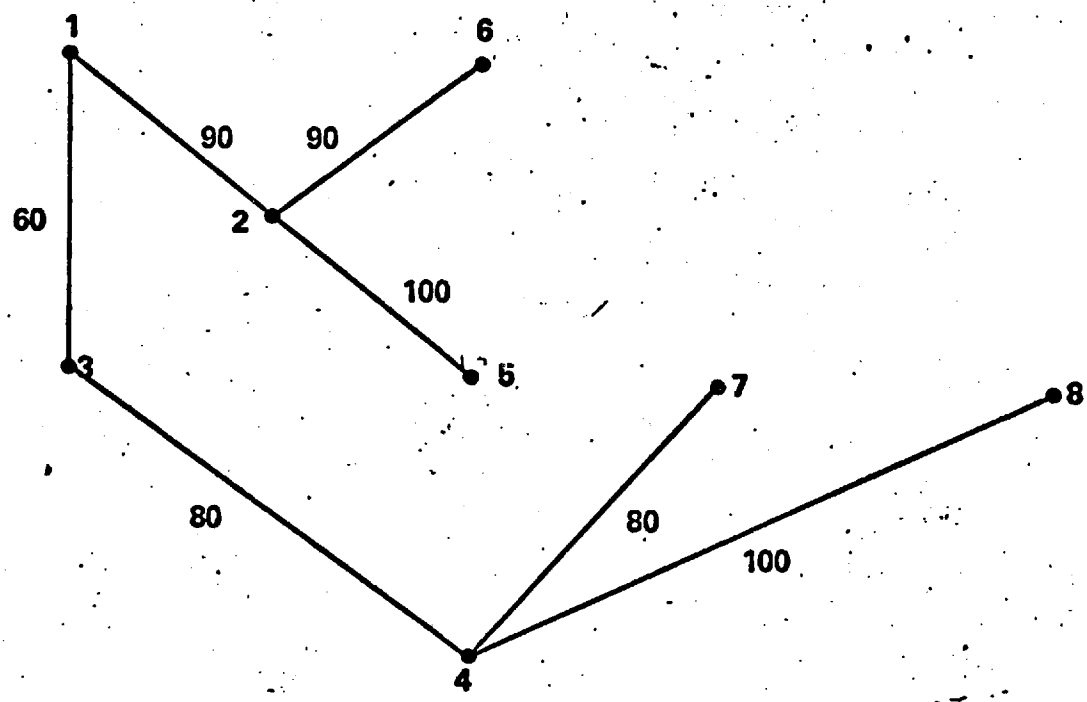

FIG 3. LONGEST SPANNING TREE FOR EXAMPLE

this becoming the new "best answer so far". Backtracking occurs at Step 7 by changing (4, 5) to -(4, 5) in S.

Node 3. Step 4 is the next active step which produces a bound < z*. The tests of Step 5 are also unsuccessful so we continue to the augmentation of Step 6.

Node 4. The smallest arc in the updated longest spanning tree is now (3, 4), (link (4, 5) having been set to infinite cost). After augmentation with (3, 4), the forced augmentations by Step 6 of -(2, 5), -(4, 8) and -(2, 6) occur. G'_S is now of size greater than four so backtracking occurs at Step 7 by setting (3, 4) to -(3, 4) in S. (Note that -(2, 5), -(4, 8) and -(2, 6) were all marked).

Node 5. Proceeding to Step 4, the calculated bound is still 160 < z* (= 230), and in Step 5 the tests were unsuccessful.

Node 6. Step 6 chooses link (1, 2) for the new augmentation, which forces further augmentations of S by -(2, 5) and -(4, 8). The graph G'_S is now size infeasible so backtracking can occur.

Continuing in the same way, we obtain the complete solution tree shown in Fig. 2, with the optimal partition being given by the vertex sets Y = {1, 2, 5, 6} and $\bar{Y}$ = {3, 4, 7, 8}. The corresponding cut is shown dotted in Fig. 1.

2.5 Computational Results

The algorithm was tested on a total of about 300 graphs, including planar, non-planar and regular graphs with varying degrees of density and varying types of link cost structures. For each given size and type of graph, 10 graphs were tested with link costs randomly generated from the same cost distribution and computing times are given for the maximum, minimum and average solution time.

From the tables which follow, it can be seen that, in general:

(i) Planarity does not affect solution times in any significant way.

(ii) Regular graphs are only slightly harder to solve than irregular ones.

(iii) Graphs whose link costs are all the same are solved in about the same time as those with uniform link cost distribution.

(iv) Solution times increase rapidly with increasing graph density, and the density is by far the most critical parameter as far as computation times are concerned.

Tables 1, 2 and 3 show, respectively, the performance of the algorithm on randomly generated irregular graphs with average vertex degrees of 3, 4 and for complete graphs. The graphs are produced by first randomly generating a spanning tree on n vertices, and then adding links to the tree at random, until the required average graph density is reached. The link costs for the graphs of tables 1, 2 and 3 are generated from a random uniform distribution between the values 1 and 10. It is

TABLE 1

COMPUTATIONAL RESULTS FOR GRAPHS WITH AVERAGE VERTEX DEGREE 3

Number of vertices	Computing times (CDC 6600 sec.)			Average number of flow calculations
	Shortest time	Average of ten graphs	Longest time	
12	0.08	0.27	0.71	462
16	0.73	1.36	3.24	275
20	0.38	3.86	15.48	429
24	2.15	11.69	20.33	970
30	12.21	26.42	35.92	1093
36	27.12	50.82	201.61	1946
40	60.12	108.30	300.00	3809
46	-	274.1†	-	5321

† One try only

TABLE 2

COMPUTATIONAL RESULTS FOR GRAPHS WITH AVERAGE VERTEX DEGREE 4

	Computing times (CDC 6600 sec.)			Average number of flow calculations
Number of vertices	Shortest time	Average of ten graphs	Longest time	
12	0.93	1.73	4.28	250
16	2.30	6.53	51.03	1760
20	9.31	19.61	127.40	1182
24	15.07	40.81	175.90	2091
30	-	233.5†	-	12504
36	-	311.1†	-	8241

† One try only

TABLE 3

COMPUTATIONAL RESULTS FOR COMPLETE GRAPHS

Number of vertices	Computing times (CDC 6600 sec.)			Average number of flow calculations
	Shortest time	Average of ten graphs	Longest time	
8	1.19	1.91	3.74	388
10	3.68	5.41	8.89	1327
12	6.14	40.18	291.1	3171
14	-	79.72†	-	4372
16	-	293.1 †	-	6845

† One try only

TABLE 4

COMPUTATIONAL RESULTS FOR GRAPHS WITH AVERAGE VERTEX DEGREE 3 AND CONSTANT LINK COSTS

	Computing times (CDC 6600 sec.)			Average number of flow calculations
Number of vertices	Shortest time	Average of ten graphs	Longest time	
12	0.23	0.36	1.17	34
16	0.37	2.13	5.74	227
20	1.15	4.12	29.01	364
24	2.01	10.41	34.97	1060
30	4.76	28.74	75.48	1900
36	10.00	77.60	250.80	2564
40	-	146.1†	-	4614
46	-	301.1†	-	8113

† One try only

seen from these tables that the algorithm is efficient for partitioning sparse graphs, but that its performance deteriorated very rapidly for average vertex degrees greater than 4. For such dense graphs, a better tree search algorithm would have been one in which the branchings would correspond to decisions as to whether a vertex of the graph is on one side of the partition or the other, rather than correspond to decisions about the links lying on the cut. However, graphs found in practice are sparse - for example, in the problem of area delineation, discussed at the beginning of this chapter, all vertex degrees are $\leq$ - and it is for such sparse graphs that the present algorithm has been developed.

Correlation tests between computing times T and number of vertices n in the graphs were performed for the results shown in Tables 1 and 2 and the following curves were fitted to the data.

For average degree 3: $T = 1.45 \times 10^{-6} \times n^{4.93}$ CDC 6600 seconds with correlation coefficient of 0.997, and for average degree 4: $T = 7.31 \times 10^{-6} \times n^{4.96}$ CDC 6600 seconds with correlation coefficient of 0.993. The dependence of computing times on number of vertices being very nearly according to a 5^{th} power for both cases.

A comparison of planar and non-planar graphs with average vertex degrees 3 and 4 was made. Planar graphs were generated as mentioned earlier (with their planarity tested manually after generation), and non-planar graphs were generated by starting with a $K_{3,3}$ Kuratowski graph as a subgraph. No computational difference between these two types of graphs was found, although computing times for

planar graphs could be reduced further, had a special maximum flow algorithm applicable to such graphs been used (16).

Tests between regular and random graphs were also performed on a set of 10 problems, and regular graphs were found to require on average approximately 20% more computing time than random ones. One 40-vertex regular graph - with two sets of cost structures - which was used in the tests is given in Fig. 4 and Table 5. This test graph was optimally partitioned into two equal parts by the present algorithm (as shown in Fig. 4) in approximately 120 secs.

Tests were also performed on the graphs used to derive Table 1, but with all link costs set to 1, rather than being randomly generated. The results are shown in Table 4 from which it is apparent that there is no significant difference between the average computing times for these graphs and for those with uniformly distributed link costs. However, the spread of computing times about the average is seen to be greater for the constant link cost case. The correlation test on the results of Table 4 gave the fit as:

$$T = 1.79 \times 10^{-6} \times n^{4.90} \text{ secs.}$$

with correlation coefficient of 0.980.

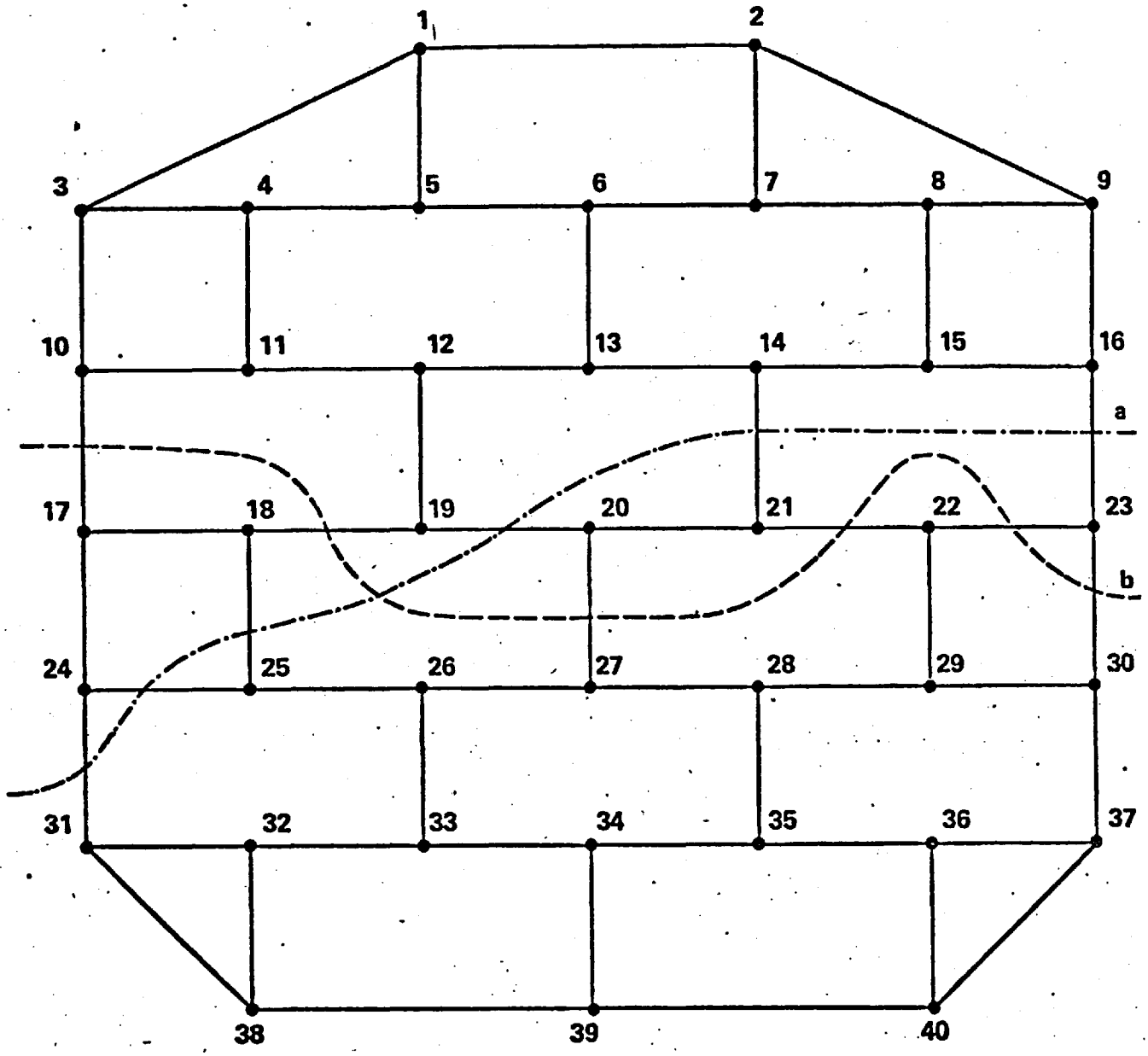

FIG 4. TEST GRAPH

TABLE 5

LINK COSTS FOR TEST GRAPH

Link	Cost		Link	Cost	
	graph a	graph b		graph a	graph b
(1,2)	10	7	(19,20)	1	1
(1,3)	4	8	(20,21)	4	8
(1,5)	2	7	(20,27)	8	2
(2,7)	6	5	(21,22)	9	4
(2,9)	8	1	(22,23)	9	1
(3,4)	5	2	(22,29)	9	7
(3,10)	2	8	(23,30)	3	7
(4,5)	9	7	(24,25)	8	8
(4,11)	8	6	(24,31)	2	8
(5,6)	1	8	(25,26)	7	10
(6,7)	4	10	(26,27)	3	10
(6,13)	10	8	(26,33)	8	6
(7,8)	3	9	(27,28)	9	5
(8,9)	5	4	(28,29)	2	4
(8,15)	2	6	(28,35)	7	5
(9,16)	4	5	(29,30)	4	9
(10,11)	8	1	(30,37)	9	8
(10,17)	6	3	(31,32)	5	5
(11,12)	3	3	(31,38)	3	10
(12,13)	2	7	(32,33)	9	4
(12,19)	3	4	(32,38)	3	2
(13,14)	7	10	(33,34)	9	2
(14,15)	5	5	(34,35)	1	2
(14,21)	3	3	(34,39)	5	2
(15,16)	4	10	(35,36)	3	8
(16,23)	3	6	(36,37)	3	3
(17,18)	4	5	(36,40)	9	10
(17,24)	6	5	(37,40)	8	9
(18,19)	6	2	(38,39)	6	4
(18,25)	2	3	(39,40)	5	9

REFERENCES

1. G. KRON. "Tensor analysis of networks". John Wiley, New York, 1939.

2. G. KRON. "Diakoptics - The piecewise solution of large-scale systems". Macdonald, London, 1963.

3. G. KRON. "Improved procedures for interconnecting piecewise solutions". Journal of the Franklin Institute, CCLXII-6, pp. 385-92, 1956.

4. H.H. HAPP. "Diakoptics and networks". Academic Press, New York, 1971.

5. B.W. KERNIGHAN and S. LIN. "An efficient heuristic procedure for partitioning graphs". Bell Systems Technical Journal, 49, pp. 291-307, 1970.

6. L.L. GORINSHTEYN. "Partitioning of graphs". Engineering Cybernetics, pp. 76-82, 1969.

7. B.W. KERNIGHAN. "Optimal sequential partitions of graphs". Journal ACM, 18, pp. 34-40, 1971.

8. R.E. GOMORY and T.C. HU. "Multi-terminal network flows". Journal of SIAM, 9, pp. 551-570, 1961.

9. L.R. FORD and D.R. FULKERSON. "Flows in networks". Princeton University Press, 1962.

10. MA.A BREUER. "Recent developments in the automated design and analysis of digital systems". Proc. IEEE, 60, pp. 12-27, 1972.

11. A. GEOFFRION. "Integer programming by implicit enumeration and Balas' method". SIAM Rev. 9, pp. 178-189.

12. P.C. GILMORE and R.E. GOMORY. "Multi-stage cutting stock problems of two and more dimensions". Ops. Res., 13, pp. 94-120, 1965.

13. P.C. GILMORE and R.E. GOMORY. "Theory and computation of knapsack functions". Ops. Res., 14, pp. 1045 -1074, 1966.

14. R.C. PRIM. "Shortest connection networks and some generalisations". Bell Systems Technical Journal, 36, pp. 1389-1401, 1957.

15. J.D.C. LITTLE, K.G. MURTY, D.W. SWEENEY and C. KAREL. "An algorithm for the travelling salesman problem". Ops. Res., 11, pp. 972-989, 1963.

16. C. BERGE and A. GHOUILA-HOURI. "Programming, games and transportation networks". Methuen, London, 1965.

17. R.S. GARFINKEL and G.L. NEMHAUSER. "Optimal political districting by implicit enumeration techniques". Man. Sci., 16, B495-B508, 1970.

18. N. CHRISTOFIDES. "Fixed routes and areas for vehicle scheduling". International Journal of Physical Distribution, pp. 87-92, Feb. 1971.

C H A P T E R I I I

OPTIMAL EXPANSION OF AN EXISTING NETWORK

3.1 The Reasons for Augmenting Networks

Physical distribution systems, such as electricity or gas supply, and road or rail networks etc., can be naturally considered as networks of flow. A common feature of these systems is that expansion or contraction may be necessary with the elapse of time, as supply and demand requirements change. In the case of an electricity power network (1), (2), let us consider the power flow from a specific source vertex s to a sink vertex t and assume that the power requirement of the sink is increased from some value P_1 to another value P_2. The existing network may not be able to accommodate this increased power flow from s to t and what may be required is to add extra links to the network to make such a flow possible. Links can be added to the network at a specific cost and capacity, and the problem is then one of choosing which links to add so as to increase the maximum flow capacity (from s to t) to a value at least as large as P_2 and at the same time incur the least cost. Alternatively, the problem may be one of choosing a set of links to add, whose total cost is less than a given budget and which causes the largest increase in the s to t flow capacity of the network.

The problem considered in this chapter can be stated as follows.

A network $G_o = (X, A)$ is given where X is a set of vertices and A a set of links of known capacity. Also

given is a set E of ordered vertex pairs (x_i, x_j) between which vertices - x_i and x_j - a link (x_i, x_j) can be added to G_o at one of m possible levels of capacity, say q_{ij}^k (k = 1, 2, ..., m) and with corresponding levels of cost given by c_{ij}^k. The problem is then one of choosing (from E) the set of links (and their level) to be added to G_o so as to maximise the largest flow possible between the two given vertices s and t, subject to a restriction that the total cost of the added links must be less than a given budget B.

A continuous version of this problem was considered by Fulkerson (4) and Hu (5), in which the assumption is made that the cost of increasing the existing link capacities is a linear function and that the increases can be made in arbitrarily small amounts. This assumption means that the problem can be formulated as a linear program. Fulkerson's paper shows that there is an algorithm which produces solutions to the problem - not only for a fixed budget, but for all budgets, i.e. the problem is solved parametrically. The algorithm is a variant of the labelling procedure due to Ford and Fulkerson (3) for the solution of maximal network flow problems and minimal cost transportation problems.

A point of interest is that, although this budget problem is not of the usual transportation form, it can be solved by a labelling procedure. Fulkerson explains that this is because, for a given budget problem, a pair of transportation-type linear programs exist such that an optimal solution to the budget problem is given by a convex combination of certain optimal solutions to the two

auxiliary problems. Thus the algorithm is designed to solve, in an efficient fashion, a sequence of related transportation-type problems - the sequence having the property that adjacent pairs of solutions produced by the algorithm can be used to generate a solution of this parametric budget problem.

In contrast, we are dealing with a combinatorial problem, in which new links can be added at one of m discrete levels of capacity and cost, or not added at all. It should also be noted that the restriction to linear cost functions is not a requirement in the present model, nor need there be any specific functional relation between link costs and capacities.

3.2 The Method

The problem as defined in the previous section could, in principle, be solved by choosing - subject to the budgetary constraint - every combination of possible links that can be added (and at what level), and for each such combination performing a maximum flow calculation from s to t. That combination of links and levels for which this maximum flow is largest would then be the solution to the problem. Needless to say, this is not an approach that could be applied to any but the smallest problem. For example, a problem in which links could be added in 10 possible places at one of three different levels of capacity would require of the order of 10^6 ($\simeq 4^{10}$) flow calculations, the exact number depending on the value of the available budget and on the link costs.

The basic method described in this section is a

tree search technique in which combinations are implicitly excluded from consideration by the use of tight upper bounds and necessary conditions that these combinations have to satisfy in order to be candidates for the optimal solution.

3.2.1 Some necessary conditions for improved solutions

Let us examine the maximum flow/minimum cut theorem which states that for a network G the maximum flow from source s to sink t is equal to the minimum cut capacity of all cuts separating s and t - where by a cut K is meant a partitioning of the vertices of G into two disjoint sets $(Y, \bar{Y})$, such that $s \varepsilon Y$, $t \varepsilon \bar{Y}$ and $Y \cup \bar{Y} = X$ the total set of vertices. The capacity of this cut is denoted by

$$q(K, G) \equiv \sum_{x_i \varepsilon Y,\ x_j \varepsilon \bar{Y}} q_{ij},$$

where q_{ij} is the capacity of the link (x_i, x_j) in G. It is an obvious corollary of the above that an increased flow between s and t can only be achieved by the addition of a link or links which <u>span</u> the minimum cut between s and t. A link (x_i, x_j) is said to span a cut $K = (Y, \bar{Y})$ if $x_i \varepsilon Y$ and $x_j \varepsilon \bar{Y}$. This however is a necessary, but not a sufficient condition for an increased flow. In particular, suppose that links $(x_{i_1}, x_{j_1})^{k_1}$, $(x_{i_2}, x_{j_2})^{k_2}$, ... $(x_{i_r}, x_{j_r})^{k_r}$ have been chosen (in that sequence), from the set of vertex pairs E with corresponding capacity levels $k_1, k_2, .., k_r$. Let these links represent the best combination found so far, (this will be referred to as a

solution), and define the graphs:

$$G_1^* = (X, A_1) \text{ where } A_1 = AU\{(x_{i_1}, x_{j_1})^{k_1}\},$$

$$G_2^* = (X, A_2) \text{ where } A_2 = A_1U\{(x_{i_2}, x_{j_2})^{k_2}\},$$

" " " " " " "

and

$$G_r^* = (X, A_r) \text{ where } A_r = A_{r-1}U\{(x_{i_r}, x_{j_r})^{k_r}\}.$$

If now the set of links A_s were added to G_o, an improvement to the value of the currently best solution - obtained by the addition of $\{(x_{i_\alpha}, x_{j_\alpha})^{k_\alpha} | \alpha = 1, .., r\}$ to G_o - would only be possible if each one of the minimal cuts K_o, K_1, .., K_r of the corresponding graphs G_o, G_1^*, .., G_r^* were spanned by at least one link of A_s. However, an arbitrarily chosen set of links A_s which spans the set of cuts K_o, K_1, .., K_r need not cause an improved flow from s to t, since the spanning of these cuts does not imply that all other cuts of capacity less than that of K_r are also spanned. Thus, the necessary condition, which A_s must satisfy for an improvemtnt to the currently best solution to be possible, can be made stricter by requiring A_s to span extra cuts, say, K_{r+1}, ..., K_f of value less than that of K_r. The algorithm described in a later section gives one possible procedure for choosing such extra cuts.

As mentioned earlier, this algorithm is a tree-search method during which the current state of the search is represented by an ordered sequence S of links on which a definite decision for inclusion or exclusion from G_o has already been made. A link $(x_i, x_j)^k$ is added positively into S if link (x_i, x_k) at capacity level k is included

into G_o, and added negatively into S (i.e. $-(x_i, x_j)^k \varepsilon S$) if (x_i, x_j) at level k is excluded from G_o. All links not in S are "free" in the sense that no decision about them had yet been made, except that if $(x_i, x_j)^k \varepsilon S$ then $(x_i, x_j)^{k'}$, for $k' \neq k$, cannot be in S, since link (x_i, x_j) has already been chosen to be at capacity level k. We will use S^+ and S^- to mean the (unordered) set of links entered into S as positive and negative entries respectively. The structure of the search itself follows that given by Goeffrion (6).

3.2.2 Bounding procedures for limiting the search

Let κ be the current set of all cuts which an improved solution is required to span. (This complete set is not available, but it is updated by the algorithm as and when it is required). Let us suppose that the set of links S^+ - spanning all cuts in $\kappa \equiv \{K_o, K, .., K_r, K_f\}$ - has already been chosen and added into the network G_o to produce the network $G(S^+)$. The cost of the links already chosen is

$$C(S^+) = \sum_{(x_i, x_j)^k \varepsilon S^+} c_{ij}^k$$

and there is a budget of $B(S^+) = B - C(S^+)$ remaining. The capacity $q\{K_\lambda, G(S^+)\}$ of a cut $K_\lambda \varepsilon \kappa$ of the network $G(S^+)$ may be below the value z* of the best flow so far, i.e. below $z^* \equiv q(K_r, G_r^*)$, this being the capacity of cut K_r for the network G_r^* defined earlier. The remaining budget must be spent adding further links - into $G(S^+)$ - which span K_λ, until $q\{K_\lambda, G(S^+)\}$ is greater than z*. If this

is not possible then no completion of the set S (i.e. no completion of decisions already made about links) could cause an improvement to the value of the best answer so far. Let E(S) be the vertex pairs of G_o across which a link can be placed given the decisions implied by S. Thus:

$$E(S) = E - \{(x_i, x_j) \mid (x_i, x_j)^k \varepsilon S^+ \text{ for some } k = 1, \ldots m\}.$$

Then let:

$$R_\lambda = \{(x_i, x_j) \mid x_i, x_j \varepsilon E(S) \text{ and spanning } K_\lambda\}.$$

Now suppose that the whole of the available budget $B(S^+)$ is to be spent choosing from R_λ those links and their capacity levels which when added to $G(S^+)$ will maximise the capacity across cut K_λ. If this maximum possible capacity is not greater than z*, the present solution cannot be improved. The same would hold true if the available budget were spent maximising the capacity across any other cut $K_\mu \varepsilon K$ and the capacity of K_μ was still less than z*.

A procedure for choosing - for a given budget $B(S^+)$ - that combination of links which maximises the capacity across a cut K_λ could then be used to provide an upper bound on the value of the maximum s-to-t flow achievable - given the set S of decisions that have already been made.

3.2.3 A bounding test†

The general maximisation problem mentioned above can be solved very effectively using a dynamic programming algorithm. For a given cut $K_\lambda \varepsilon K$ let the elements of R_λ be numbered from 1 to p, where a link across a vertex pair can be chosen at one of m levels. (It will be assumed

† See the Appendix to this chapter for a detailed example.

here that the costs c_{ij}^k for all links $(x_i, x_j)^k \varepsilon S^-$ have been set to infinity which implies that link (x_i, x_j) can never be chosen at level k).

Now let $f_\ell(b)$ be the maximum total capacity of links that can be chosen given a budget b and using only links from the first ℓ vertex pairs of R_λ. The functions $f_\ell(b)$ ($\ell = 1, \ldots, p$) can then be calculated iteratively from the equation

$$f_\ell(b) = \underset{(k \mid c_{\alpha\beta}^k \leq b)}{\text{Maximum}} \{q_{\alpha\beta}^k + f_{\ell-1}(b - c_{\alpha\beta}^k)\}, \quad (1)$$

where (x_α, x_β) is the ℓ^{th} vertex pair of R_λ. Equation (1) can be initialised by $f_o(b) = 0$ for all b such that $0 \leq b \leq B(S^+)$. The value of the function $f_p\{B(S^+)\}$ is then the maximum possible increase to the capacity of cut K_λ, and if

$$q\{K_\lambda, G(S^+)\} + f_p\{B(S^+)\} \leq z^* \quad (2)$$

for any cut K_λ, then a backtracking step on the set S can be taken. (See description of the algorithm in section 3.3).

The bounding test of equation (2) investigates each cut $K_\lambda \varepsilon K$ individually and determines whether the capacity of this cut could conceivably become greater than z*, when the total remaining budget $B(S^+)$ was spent on spanning it. However, a better solution could only result when the capacities of all cuts in K become greater than z* simultaneously. Thus a better bounding test can be derived as follows. Because a dynamic programming algorithm has been used to calculate $f_p\{B(S^+)\}$ from equation (1), the values of all $f_p(b)$ for $0 \leq b \leq B(S^+)$ are available at the

end of the computations. Let b_λ be the smallest value of b such that

$$q\{K_\lambda, G(S^+)\} + f_p(b) > z^* \qquad (3)$$

Such a value for b_λ $\{0 \leq b_\lambda \leq B(S^+)\}$ would always exist otherwise the test of equation (2) would not have failed. The value of b_λ is the smallest budget that must be spent for the capacity across cut K_λ to become greater than the value of the best flow so far.

Now suppose that another cut $K_\mu \varepsilon K$ is "disjoint" with respect to K_λ in the sense that $R_\lambda \cap R_\mu = \Phi$, (i.e. no links span both cuts) and suppose that b_μ is defined in the same way as b_λ. We must now have $b_\lambda + b_\mu \leq B(S^+)$ otherwise the capacities across cuts K_λ and K_μ cannot both be made more than $q(K_r, G_r^*)$ and a backtracking step on the set S can be taken.

In general, if there are h "disjoint" cuts $K_\rho \varepsilon K$, ($\rho = \rho_1$ to ρ_h), we must have

$$\sum_{i=1}^{h} b_{\rho_i} \leq B(S^+) \qquad (4)$$

A tighter test than (4), which also considers cuts which are not disjoint in the above mentioned sense, is possible at a considerable increase in computational cost.

3.2.4 An augmentation rule

During the tree-search, the situation arises that a new link is to be chosen for addition into the network (at some level), in which case the set S, defined earlier as the set of links about which a decision has already been made, must be augmented by the newly chosen link.

Consider two nodes in the tree, one represented by $N_1 = SU\{(x_i, x_j)^k\}$ and the other by $N_2 = SU\{-(x_i, x_j)^k\}$, i.e. node N_1 corresponds to adding link $(x_i, x_j)^k$ into the network (in addition to the links already in S) and node N_2 corresponds to excluding $(x_i, x_j)^k$ from the network. In general, the complete subtree following node N_2 is larger than the subtree following N_1, because there is more budget available at N_2 and hence a greater choice of branching from that node. Thus, the policy was adopted of choosing that augmenting link $(x_i, x_j)^k$ which would tend to minimise the size of the subtree following node N_2, rather than produce the best possible flow corresponding to node N_1. This augmenting policy of choosing the link with the highest "apparent opportunity cost" is similar in spirit to that used by Little et al (9) for the travelling salesman problem.†

The method used for choosing an augmenting link is then as follows. Let us suppose that the set S already spans all the cuts in κ as required and that a budget $B(S^+)$ remains for spending. The minimum cut, K_ν say of the graph $G(S^+)$ is now calculated. (As mentioned earlier, the next link to be chosen must span this cut if the maximum flow value is to increase). The maximum increase in the value of the flow achievable by spending the budget $B(S^+)$ can now be calculated by the iterative procedure of expression (1), as was done previously for the cuts in κ. In fact, it should be noted that for the majority of augmentations it so happens that $K_\nu \varepsilon \kappa$. Moreover, since an augmentation of S will only take place, if the bounding tests of expressions (2) and (3) have failed for all cuts in κ,

---

† See section 1.5 for a brief discussion of this.

then at the time when an augmentation is required, the results of the computation of expression (1) already exist for all cuts in κ, and hence in these cases a new calculation for cut K_ν will not be required.

Now suppose that Δ is the set of links - with total cost $C(\Delta)$ - in the optimal dynamic programming solution corresponding to cut K_ν, and that $(x_{i_o}, x_{j_o})^{k_o} \in \Delta$. Had link $(x_{i_o}, x_{j_o})^{k_o}$ not been in the solution, the best way it could have been replaced would be by another link $(x_{i'}, x_{j'})^{k'}$ where:

$$q^{k'}_{i'j'} = \underset{(x_i, x_j)^k \in R_\nu,\ \notin \Delta}{\text{Maximum}} \{q^k_{ij} \mid c^k_{ij} \leq B(S^+) - C(\Delta) + c^{k_o}_{i_o j_o}\} \quad (5)$$

The assumption made in equation (5) is that only link $(x_{i_o}, j_o)^{k_o}$ is being replaced and that the rest of the links in the optimal solution Δ remain the same. The (non-negative) difference $\delta = (q^{k_o}_{i_o j_o} - q^{k'}_{i'j'})$ is taken as a measure of the importance of link $(x_{i_o}, x_{j_o})^{k_o}$. This is because the omission of that link in Δ with the largest value of δ would lead to a node in the tree from which an early backtracking can be expected due to the subsequent forward branchings failing the bounding tests. Ties in the values of δ can be broken arbitrarily.

3.3 A Description of the Algorithm

The basic algorithm will now be described, and together with the comments given in the last section the algorithm should be self-explanatory.

Initialisation

Step 0 Set $S \leftarrow \Phi$, $\kappa \leftarrow \Phi$, $B(S^+) = B$, S^* (the

best solution so far) = Φ and z^* (value of S^* maximum flow) = 0.

Step 1 — Perform an s to t flow calculation on the network $G(S^+)$. Let z be the value of the maximum flow and K_{MAX} the corresponding cut.

Step 2 — Set $\kappa = \kappa U(K_{MAX})$, if $z \geq z^*$ set $S^* \leftarrow S$, $z^* \leftarrow z$.

Bounding Test on K_{MAX}

Step 3(i) — Calculate the vector $f_p(q)$, $0 \leq q \leq B(S^+)$, for the cut K_{MAX} using expression (1) and perform the bounding tests (2) and (3). (The vector $f_p(q)$ may already have been calculated at a previous pass). If the tests are successful go to step 4, else go to step 7.

Bounding Test on K

Step 3(ii) — Calculate the vectors $f_p(q)$, $0 \leq q \leq B(S^+)$ for each cut in κ from equation (1), and perform the bounding tests (2) and (3). If either test is successful, go to step 4, else go to step 1.

Backtracking

Step 4 — Negate the last positive entry - say $(x_i, x_j)^k$ - in S and remove all entries following it. Reset the costs c_{ij}^{ℓ} for all $\ell = 1, .., m$ back to their original values. Go to

step 5.

If there are no positive entries left, stop. The set S* is the optimal solution and z* is its value.

Block Augmentation to Satisfy Necessary Conditions

Step 5 — If all cuts in κ are spanned by links in S^+, go to step 3(ii), else to step 6.

Step 6 — Add links $(x_i, x_j)^k \notin S^+$ and $\notin S^-$ into set S in ascending order of their magnitude of cost c_{ij}^k until either all cuts in κ are spanned, or an unspanned cut in κ is found which, (given the remaining budget), cannot be spanned by any link. In the former case S satisfies all the necessary spanning conditions and one proceeds to step 3(ii). In the latter case, go to step 4.

Augmentation of S

Step 7 — Choose that link $(x_i, x_j)^k$ with the greatest value of δ calculated according to expression (5). Append $(x_i, x_j)^k$ as the last entry of S. Go to step 3(ii).

3.4 An Example

The algorithm described in the last section will now be used to work out the first part of the tree search for the example given in Fig. 1. In this figure, vertex 1

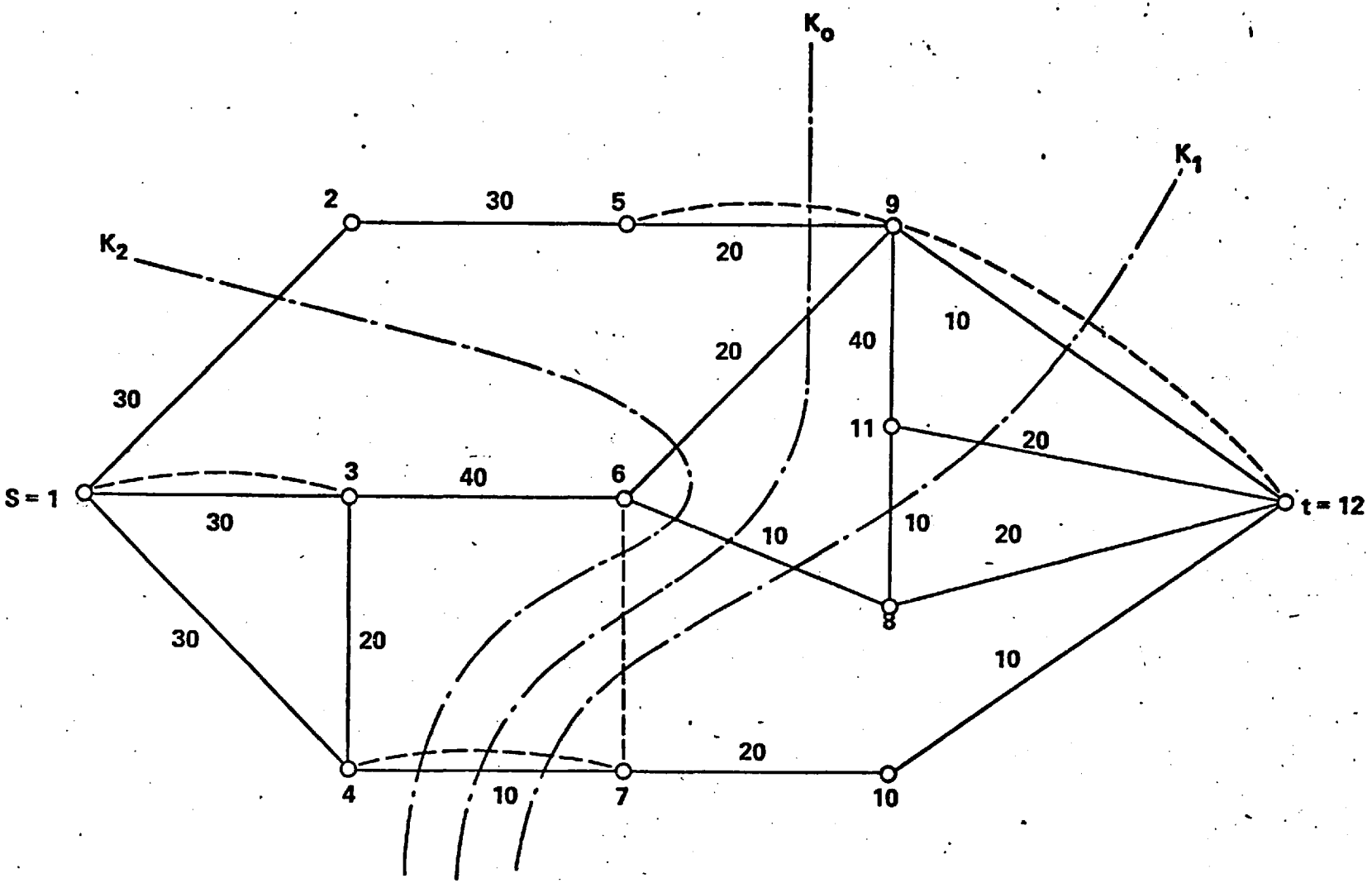

FIG 1. NETWORK FOR THE EXAMPLE

is taken as the source and vertex 12 as the sink.. The heavy lines represent the (undirected) links in the original network and the attached numbers represent the link capacities. The dotted lines represent vertex pairs across which new links can be added at one of two possible levels of capacity, i.e. 10 or 20 units. The cost of these possible additional links is given in Table 1 below. The available budget is 10 units.

Table 1. Cost Matrix (c_{ij}^k)

Vertex Pairs	(4,7)	(5,9)	(9,12)	(1,3)	(6,7)
k=1, Link at capacity 10	2	2	3	1	5
k=2, Link at capacity 20	8	9	6	3	7

The search tree is shown in Fig. 2 and the algorithm proceeds as follows:

Node 0: Step 0: Initialise variables $S = S^* = \kappa = \Phi$, $B(S^+) = 10$, $z^* = 0$.

Step 1: s to t flow calculation is performed giving $z = 60$ and $K_{MAX} = K_o = \{(1,2,3,4,5,6),(7,8,9,10,11,12)\}$.

Step 2: z^* set to 60, $\kappa = \kappa U\{K_o\} = \{K_o\}$.

Step 3(i): The vector $f_p(q)$ {in this case p=3 vertex pairs, i.e. (5,9), (6,7), and (4,7)} for $0 \leq q \leq 10$ is calculated. The bounding test (2) fails. The maximum increase in the capacity of cut K_o which is possible for a budget of 10 units is $f_3(10) = 30$, and

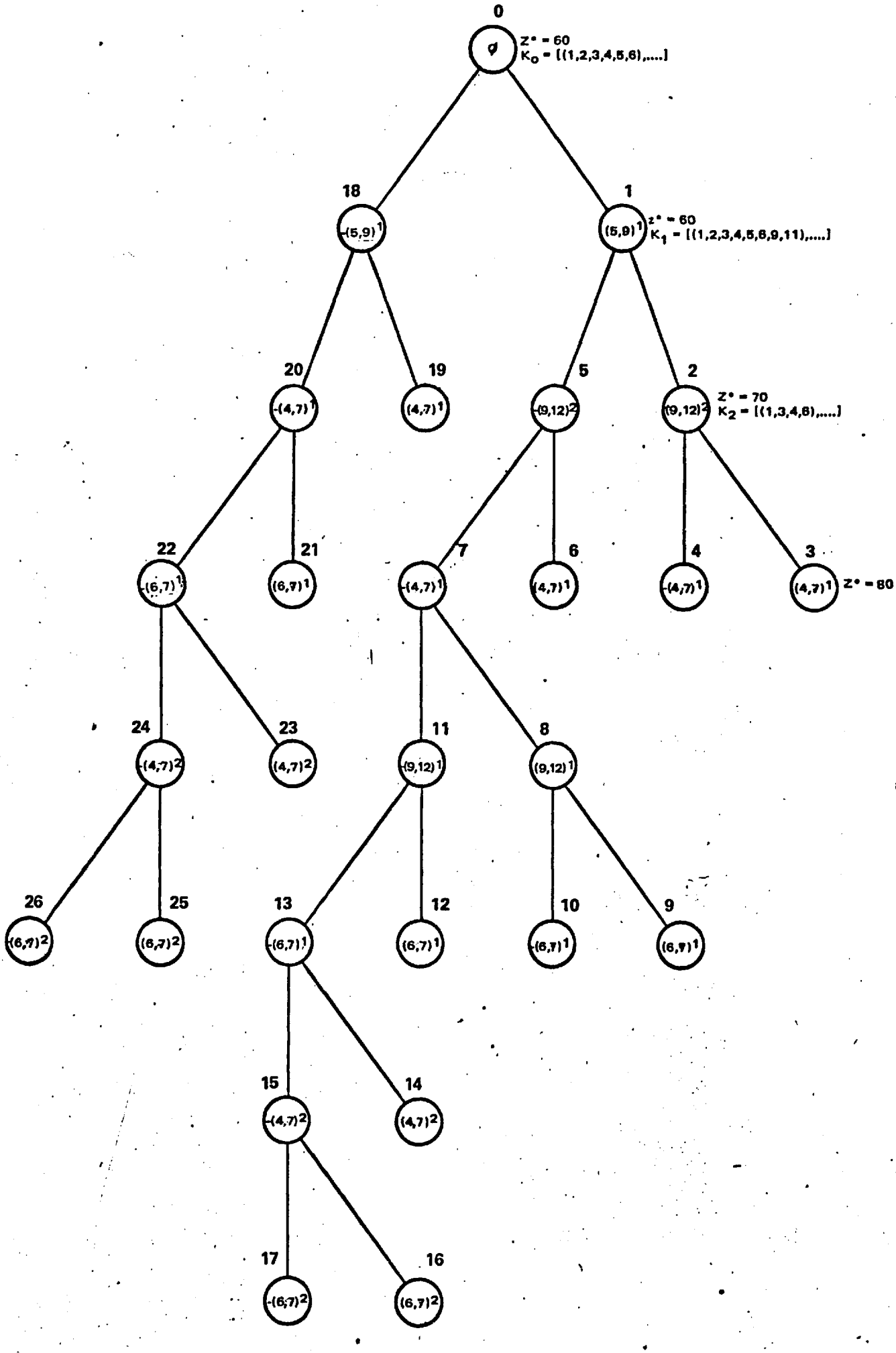

FIG 2. THE COMPLETE TREE SEARCH FOR THE EXAMPLE

this is achieved by the choice of links $(4,7)^2$ and $(5,9)'$, where the superscript refers to the level K at which the link is added.†

Node 1: Step 7: Link $(5,9)'$ is chosen for augmentation of the set S by the δ criterion {$\delta = 10$ for this link, $\delta = 0$ for $(4,7)^2$}. S is now $\{(5,9)'\}$.

Step 3(ii): Bounding test on $\kappa = (K_o)$ fails.

Step 1: s to t flow calculation is performed once more giving $z = 60$, but $K_{MAX} = K_1 = \{(1,2,3,4,5,6,9,11), (7,8,10,12)\}$.

Step 2: $\kappa = \kappa U(K_1) = (K_o, K_1)$.

Step 3(i): The vector $f_p(q)$ (in this case $0 \leq q \leq B(S^+) = 8$, and $p = 3$ i.e. the vertex pairs (4,7), (9,12) and (6,7) spanning K_{MAX}) is calculated and test (2) fails ; links in the optimal dynamic programming solution are $(4,7)'$ and $(9,12)^2$.

Node 2: Step 7: Link $(9,12)^2$ is chosen for augmenting S, both $(9,12)^2$ and $(4,7)'$ having δ equal to 10. $B(S^+)$ is now 2, and $S = \{(5,9)', (9,12)^2\}$.

Step 3(ii): Bounding test on $\kappa = (K_o, K_1)$ fails.

Step 1: Flow calculation gives $z = 70$ and $K_{MAX} = K_2 = \{(1,3,4,6), (2,5,7,8,9, 10,11,12)\}$.

Step 2: $\kappa = \kappa U K_2 = (K_o, K_1, K_2)$ and $z^* = 70$.

---

† This capacity could be achieved using other links - e.g. $(5,9)'$, $(6,7)'$, $(4,7)'$ with cost 9.

	Step 3(i):	The test on K_{MAX} fails with the optimal dynamic programming solution being (4,7)'.
Node 3:	Step 7:	Link (4,7)' is chosen for augmenting S, $B(S^+) = 0$, and S = {(5,9)', $(9,12)^2$,(4,7)'}.
	Step 3(ii):	Bounding test fails.
	Step 1:	s to t flow calculation gives z = 80 with K_{MAX} the same as K_o.
	Step 2:	$z^* = 80$, S is now {(5,9)',$(9,12)^2$, (4,7)'}.
	Step 3(i):	The remaining budget $B(S^+) = 0$ and test (2) is of course successful.
Node 4:	Step 4:	Backtrack, putting last positive entry in S negative. S is now {(5,9)',$(9,12)^2$,-(4,7)'}, $B(S^+) = 2$.
	Step 5:	K_2 is not spanned.
	Step 6:	No link $\notin$S (of cost less than or equal to 2) can be found to span K_2.
Node 5:	Step 4:	Backtrack to S = {(5,9)', $-(9,12)^2$}.
	Step 5:	K_1 is not spanned.
Node 6:	Step 6:	Least cost link (4,7)' is added to span K_1, S = {(5,9)',$-(9,12)^2$,(4,7)'}.
	Step 3(ii):	Bounding test (2) succeeds on cut K_1, since with $B(S^+) = 6$, we obtain $q\{K_1,G(S)\} = 70$, $f_p(6) = 10$ (i.e. adding (6,7)'), whereas $z^* = 80$.
Node 7:	Step 4:	Backtrack to S = {(5,9)',$-(9,12)^2$, -(4,7)'}.
	Step 5:	K_1 not spanned (nor is K_2).

Node 8: Step 6: Link $(9,12)'$ added to S to span K_1, $S = \{(5,9)', -(9,12)^2, -(4,7)', (9,12)'\}$.

Node 9: Step 6: Link $(6,7)'$ added to S to span K_2, $S = \{(5,9)', -(9,12)^2, -(4,7)', (9,12)', (6,7)'\}$.

Step 3(ii): Bounding test (2) succeeds on cut K_o.

Node 10: Step 4: Backtrack to $S = \{(5,9)', -(9,12)^2, -(4,7)', (9,12)', -(6,7)'\}$.

etc.

The complete tree search is shown in Fig. 2 and the solution corresponding to node 3 is the optimal one. It should be noted that by far the most time consuming operation is a flow calculation and that this does not occur at every node of the tree, but only in 5 of the total of 27 nodes, shown in Fig. 2, these five nodes being shown by heavy circles.

3.5 Computational Results

The previously described algorithm has been coded and tested on a number of problems varying in size from 20 vertices and 30 links to 60 vertices and 90 links. Additional links could be added across 20 vertex pairs for the larger problems and a link could be chosen from one of three possible levels for each vertex pair. The computer code was written in FORTRAN IV for the CDC 6600 computer and was run using the FUN compiler of that machine. The total memory requirements of the code used for the solution of each of the problems given in Table 2 was 25 K words.

In all, 10 test problems were generated as

follows: starting with the required set of n vertices, a spanning arborescence was randomly generated and link capacities were allocated from a uniform distribution. Each additional link was then randomly generated so as to span the minimum cut of the network so far, until the required number of links was added. The capacities of these links were also randomly generated from the same uniform distribution.

The vertex pairs across which new links could be added, were generated at random. For each vertex pair a link was allowed to be added at one of 3 different levels with the capacity of a link at level 3 being between 2 and 5 times that of a link at level 1. The cost-capacity relation for links at different levels across a given vertex pair was chosen to be monotonically increasing but otherwise random. For each problem the budget was chosen to be one quarter of the sum of the level 2 link costs across all the allowable vertex pairs.

Table 2 shows the results for the 10 test problems and the computational times given in columns D and E are in CDC 6600 seconds. Column G gives the number of cuts stored in the set κ during the computation. This is the number of distinct cuts (omitting replications), which explains why this number is not necessarily equal to or greater than the number of links in the final solution (which is given in column I).

Column H gives the number of flow calculations performed during the course of the algorithm. It should be noted that a flow calculation after a backtracking step, can be initialised by the maximum flow of the original

network. Moreover, after an augmentation, the labelling to find the new maximum flow can be initiated starting with the flow in the previous network and, therefore, only one or two "breakthroughs"† are needed to obtain the final answer in the new network.

The results shown in Table 2 indicate a comparatively slow increase of computational time with problem size. It is interesting to note that for a network with 20 vertex pairs across which links can be added at one of 3 levels, the total number of ways these links can be chosen is $4^{20} \approx 10^{12}$. This represents an enormous number of flow calculations, even allowing for the fact that a very large fraction of these ways are infeasible because of the budget constraint.

The effectiveness of the dynamic programming bounding tests can, to some extent, be judged by the fact that a rerun of the program without this test on problem 4 did not finish after 140 seconds and hence it is possible that these tests reduce computing times by at least a factor of 10 and possibly a great deal more.

---

† A "breakthrough" is a labelling of the sink node.

TABLE 2

	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)
Network	Number of vertices	Number of links	Number of vertex pairs*	Total time	Time to reach optimum	(E) as percentage of (D)	Number of cuts stored in K	Number of flow calculations performed	Number of links in solution
1	20	30	16	32.2	12.5	39	9	299	6
2	20	30	18	11.4	0.69	6	3	48	6
3	30	50	18	14.2	6.61	47	4	75	6
4	30	50	20	13.6	1.39	10	4	20	7
5	40	60	16	2.49	0.60	24	5	8	5
6	40	60	16	4.69	0.91	19	5	16	6
7	40	60	16	18.9	5.50	29	8	27	7
8	50	75	20	32.2	12.5	38	9	823	7
9	50	75	20	37.3	16.3	43	6	97	9
10	60	90	20	51.1	14.3	28	10	98	8

* Links can be added at one of three levels between each vertex pair

REFERENCES

1. J.C. KALTENBACH, J. PESCHON and E.H. GEHRIG. "A mathematical optimization technique for the expansion of electric power transmission systems". Trans. IEEE, PAS-89, 1, pp. 113-119, 1972.

2. L.L. GARVER. "Transmission network estimation using linear programming". Trans. IEEE, PAS-89, 7, pp. 1688-1697, 1972.

3. L.R. FORD and D.R. FULKERSON. "Flows in networks". Princeton University Press, Princeton, N.J., 1962.

4. D.R. FULKERSON. "Increasing the capacity of a network: The parametric budget problem". Man. Sci., 5, 4, pp. 472-483, 1959.

5. T.C. HU. "Integer programming and network flows". Addison-Wesley, Reading, Massachusetts, 1970.

6. A. GEOFFRION. "Integer programming by implicit enumeration and Balas' method". SIAM Rev., 9, pp. 178-189, 1966.

7. R.D. WOLLMER. "Some methods for determining the most vital link in a railway network". The RAND Corporation Report RM-3321-ISA. 1963.

8. R.D. WOLLMER. "Algorithms for targeting strikes in a lines-of-communication (LOC) network". The RAND Corporation Report RM-5864-PR. 1969.

9. J.D.C. LITTLE, K.G. MURTY, D.W. SWEENEY and C. KAREL. "An algorithm for the travelling salesman problem". Ops. Res., 11, pp. 979-989, 1963.

APPENDIX

An Example of the Bounding Test

Suppose we have p vertex pairs, pair ℓ having cost c_ℓ^k for capacity q_ℓ^k. Let $f_\ell(b)$ be the maximum total capacity of the links that can be chosen given a budget b and using only links chosen from the first ℓ vertex pairs - a vertex pair can be realised at only one capacity. The functions $f_\ell(b)$ for $\ell = 1,..,p$ can be calculated iteratively from the equation

$$f_\ell(b) = \underset{\{k \mid c_\ell^k \leq b\}}{\text{Maximum}} \{q_\ell^k + f_{\ell-1}(b - c_\ell^k)\}. \qquad \text{(A1)}$$

(This is merely equation (1) of the main text, with slightly different notation). The value of f_0 for all arguments is zero, as is the value of $f_\ell(0)$ for all ℓ.

We illustrate this method by the set of vertex pairs in table A1, in which there are three capacity levels for each of the five vertex pairs - the capacities being 10, 20 and 30. In table A2 we give the values of $f_\ell(b)$ for all values of b less than or equal to 5 units. In the bottom right-hand corners of the cells, we give the maximum values as determined by equation (A1) in the cases where $f_{\ell+1}(b) \neq f_\ell(b)$. Thus, for example, the highest capacity possible with 3 budget units, using the first two vertex pairs, is 50, this being

$$f_1(2) + q_2^2 = q_1^3 + q_2^2 = 30 + 20 = 50.$$

TABLE A1

COST MATRIX c_e^k FOR THE EXAMPLE

	COST		
vertex pair	capacity 10	capacity 20	capacity 30
1	1	2	2
2	1	1	2
3	1	2	2
4	1	1	2
5	1	2	2

TABLE A2

VALUE OF $f_\ell(b)$ FOR THE EXAMPLE

function f \ budget b	1	2	3	4	5
f_1	10 q_1^1	30 q_1^3	30 q_1^3	30 q_1^3	30 q_1^3
f_2	20 q_2^2	30	50 $f_1(2) + q_2^2$	60 $f_1(2) + q_2^3$	60 $f_1(2) + q_2^3$
f_3	20	30	50	60	80 $f_2(3) + q_3^3$
f_4	20	40 $f_3(1) + q_4^2$	50	70 $f_3(3) + q_4^2$	80
f_5	20	40	50	70	80

C H A P T E R I V

THE P-MEDIAN OF A GRAPH AND RELATED PROBLEMS

4.1 Introduction

Suppose a manufacturer decides that his goods should be stored in supply depots - his customers being supplied from the depots, rather than from the main factory. For a single depot and the customers associated with that depot, it is necessary to consider the best method of supplying the customers from the depot. The customers may be grouped in neighbourhoods, so that each group of customers will require an integral number of vehicle loads. Thus, a vehicle leaves the depot, supplies a group of customers and returns to the depot.

The customer groups can be represented as vertices of a graph and the road (or other mode of transport) network as the links. In practice, each customer group will be weighted by a number h_j representing the "importance" of that customer. Thus, for instance, h_j may be proportional to the total annual demand, or the frequency with which a vehicle has to visit that customer group in order to satisfy its demand.

A possible objective we might consider for the depot location would be to minimise the weighted sum of the total distance travelled: this location is known as the median for the graph specified by the customers. If there are to be several depots in the distribution network then we can set a similar objective of minimising total weighted distance to the nearest depot - this is the p-median of the

associated graph. These concepts will be quantified in later sections of this chapter.

The same problem appears in several forms: the location of switching centres in telephone networks, substations in electric power networks, and the location of sorting offices for letters.

4.2 The Problem

The basic reference on the p-median problem is that of Hakimi (1). Hakimi discusses the general problem of locating facilities on a network, in particular, he analyses the so-called "p-centre" problem in which p locations on the network are chosen so as to minimise the maximum distance from any vertex to the nearest facility. An algorithm for this problem is given by Christofides and Viola (2). It is, however, completely different in character to the algorithms discussed here.

For a graph G = (X,A) with link weights ℓ_{ij} (= length) with a finite number of vertices n, the (symmetric) shortest path distance between x_i and x_j is denoted by d_{ij}, this being the minimum sum of link weights for paths joining x_i and x_j. Such a distance matrix can be found using one of many efficient algorithms - for instance, that of Floyd (3). Following the definition of Hakimi, we say that a vertex x_m is a median of G if for any x_k

$$\sum_{j=1}^{n} d_{mj} \leq \sum_{j=1}^{n} d_{kj} ; \qquad (1)$$

the value of the LHS is the median length L of G. This concept can be further generalised if the vertices of G

are weighted with fixed non-negative numbers h_i ($i = 1,2,\ldots,n$) then for x_m to be a median we must have

$$\sum_{j=1}^{n} h_j d_{mj} \leq \sum_{j=1}^{n} h_j d_{kj} \tag{2}$$

for all x_k.

This is in a sense the best vertex location. One might suppose that (as in the case of centres, which we will mention again) a location on a link might give an even better location. In fact this is not the case, a proof of this being due to Hakimi (1). We will reproduce it here because of its importance and because it may give some insight into possible algorithms.

Suppose that Y is an arbitrary point on G, so that $d(Y,x_j)$ is the minimum distance to vertex x_j. We must show that there exists a vertex $x_m \in X$ satisfying

$$\sum_{j=1}^{n} h_j d(Y,x_j) \geq \sum_{j=1}^{n} h_j d_{mj} \tag{3}$$

Let Y be on the link (x_p,x_q) and renumber the vertices of G so that for $i = 1,2,\ldots,r$ the distance $d(Y,x_j)$ is measured relative to a path through x_p and for $i = (r+1), (r+2), \ldots,n$ relative to x_q, so that

$$d(Y,x_j) = \begin{cases} d(Y,x_p) + d_{pj}, & 1 \leq j \leq r \\ d(Y,x_q) + d_{qj}, & r+1 \leq j \leq n \end{cases} \tag{4}$$

Thus, we have

$$\sum_{j=1}^{n} h_j d(Y,x_j) = \sum_{j=1}^{r} h_j \{d(Y,x_p) + d_{pj}\} + \sum_{j=r+1}^{n} h_j \{d(Y,x_q) + d_{qj}\}; \tag{5}$$

We also have

$$\sum_{i=1}^{r} h_i \geq \sum_{i=r+1}^{n} h_i, \tag{6}$$

this can always be achieved by reordering. From elementary considerations it is true that

$$d(Y,x_q) = d_{pq} - d(Y,x_p), \tag{7}$$

$$d_{pq} + d_{qj} \geq d_{pj}. \tag{8}$$

Combining equations (5), (7) and (8), we get

$$\sum_{j=1}^{n} h_j d(Y,x_j) \geq \sum_{j=1}^{n} h_j d_{pj} + \{\sum_{j=1}^{r} h_j - \sum_{j=r+1}^{n} h_j\} d(Y,x_p). \tag{9}$$

which, in conjunction with the inequality (6), shows that if a point on a link is a median then so is at least one of the vertices attached to the link.

The obvious extension of the median concept is to the multimedian, which is also due to Hakimi (4). A p-median of a graph is defined as follows. Given a set of p vertices of a graph G ($1 \leq p \leq n$), that is $Y_p = x_{m_1}, x_{m_2}, \ldots, x_{m_p}$, we define the distance of a vertex x_i from Y_p as

$$d(x_i,Y_p) = \min_{k=1, \ldots, p} \{d_{im_k}\} \tag{10}$$

Then a p-median is defined as the set Y_p^* such that

$$\sum_{i=1}^{n} h_i d(x_i,Y_p^*) \leq \sum_{i=1}^{n} h_i d(x_i,Z_p) \tag{11}$$

where Z_p is any set of p points in G. It can be shown {see Hakimi (4)} that we need only consider vertex sets rather than general points on the graph G. The method used by Hakimi to determine the p-median of a graph is complete enumeration. This, while being optimal as regards the solution, is computationally very lengthy because of

the C_p^n possible vertex sets. It is interesting to note that Goldman (5) shows that the conclusions about optimal vertex location remain true under more general conditions about intervertex costs.

The problem of finding the p-median of a graph is the central question in a general class studied in the literature as "facility location and allocation" (6,7,8) and "depot location" (9,10,11,12). These problems are somewhat more general than the p-median problem in that fixed costs f_i are associated with the vertices x_i, as well as the variable costs we see in equation (11). Thus if we define the left hand side of this equation as $L(Y_p^*)$ the problem is to minimise the sum of $L(Y_p^*)$ and the sum of those f's corresponding to vertices in the set Y_p^*. In practice, the fixed costs represent the cost involved in setting up a facility at a vertex - thus the p-median problem corresponds to the case when all f_i are equal.†

Once we allow the existence of these fixed costs, it is also worthwhile considering the problem in which $|Y_p^*|$ is not required to be exactly p, but any integer less than or equal to p. We will comment in the following sections on this and the fixed cost problem with regard to the algorithmic procedures described.

There are many added restrictions which can be placed on the problem, for example, a finite number of customers for each depot, or forbidden vertices. The difficulties in solving practical depot location problems are in fact a result of the combinatorial troubles in the pure p-median problem. In the following, we will concentrate on these aspects.

---

† See the Appendix for a discussion on fixed costs.

4.3 The p-Median Problem as an Integer Program

Let (ξ_{ij}) be an allocation matrix, so that:

$\xi_{ij} = 1$ implies that vertex x_j is allocated to vertex x_i

$= 0$ otherwise.

Further, we will take $\xi_{ii} = 1$ to imply that vertex x_i is a median vertex and $\xi_{ii} = 0$ otherwise. The p-median problem can then be stated as follows:

$$\text{Minimise: } z = \sum_{i=j}^{n} \sum_{j=1}^{n} d_{ij}\, \xi_{ij} \tag{12}$$

subject to:

$$\sum_{i=1}^{n} \xi_{ij} = 1 \quad \text{for } j = 1, \ldots, n \tag{13}$$

$$\sum_{i=1}^{n} \xi_{ii} = p \tag{14}$$

$$\xi_{ij} \leq \xi_{ii} \quad \text{for all } i, j = 1, \ldots, n \tag{15}$$

and $$\xi_{ij} = 0 \text{ or } 1 \tag{16}$$

where (d_{ij}) is assumed to be the weighted distance matrix of the graph, i.e. it is the distance matrix with every column j multiplied by h_j. Equation (13) ensures that any given x_j is allocated to one and only one median vertex x_i. Equation (14) ensures that there are exactly p median vertices, and constraint (15) guarantees that $\xi_{ij} = 1$ only if $\xi_{ii} = 1$, i.e. allocations are made only to vertices that are in the median set. If $\{\bar{\xi}_{ij}\}$ is the optimal answer to the problem defined by equations (12-16) above, then the p-median is:

$$\bar{X}_p = \{x_i \mid \bar{\xi}_{ii} = 1\}$$

If constraint (16) of the above problem is replaced by:

$$\xi_{ij} \geq 0 \qquad (17)$$

then the resulting problem is a linear program (LP), whose solution can easily be obtained. {Note that an upper bound of value 1 on ξ_{ij} is not needed since $\xi_{ij} \leq 1$ is already implied by constraint (15)}.

The solution to the LP is not necessarily all integer and fractional values of ξ_{ij} can occur. ReVelle and Swain (13) report that such fractional values occur rarely, and therefore one could use the LP formulation to obtain the p-median for most problems. In any case, if fractional values of some ξ_{ij} occur, then a resolution of these uncertainties can be obtained by a tree-search procedure (13) in which one branch fixes some fractional variable ξ_{ij} to 0 and another fixes the same ξ_{ij} to 1. One can then proceed to resolve the LP's for each of the two resulting subproblems and so on until all ξ_{ij} become either 0 or 1.

An alternative approach via linear programming is given by Marsten (14), who shows that the solution $(\bar{\xi}_{ij})$ corresponding to the p-median of a graph as described by equations (12) to (16), is an extreme point of a certain polyhedron H, and that all other p-medians for $1 \leq p \leq n$ are also extreme points of H. By using Lagrange multipliers and parametric linear programming, Marsten gives a method of traversing a path among a few of the extreme points of H. This path successively generates the p-medians of the graph G in descending order of p, although certain p-medians (for some values of p) may be missed and never generated, or, conversely, extreme points of H may be generated which do not correspond to p-medians of G i.e.

contain fractional values of ξ_{ij}. Thus, although this method is both theoretically and computationally attractive, it may fail to produce the p-median of a graph for the specific value of p that may be required. Reference (14) reports the case of a complete 33-vertex graph all of whose p-medians have been successfully generated for p = 33, 32, .., 10, but whose 9-median and 8-median could not be obtained by this method.

4.4 Direct Tree Search Algorithms for the p-Median Problem

4.4.1 A direct tree search algorithm

In the previous section, we have examined the formulation of the p-median problem as an integer program. Clearly a direct tree search approach is better suited to exploiting the problem structure. Let us consider a tree search in which each subproblem generated by the branching from a tree node is produced by setting (for a given vertex x_j) a variable ξ_{ij} to unity for some vertex x_i. This implies that vertex x_j is allocated to vertex x_i: clearly x_i must be a median vertex.

The organisation of the tree search is very simple. We set up an array $M = (m_{kj})$ in which the j^{th} column contains the vertices of the graph G in ascending order of distance from vertex x_j. We illustrate this by the graph of Fig. 1(a), in which the vertices have unit weights. Obviously, the nearest vertex to vertex x_j is itself, so that $m_{1j} = x_j$. Thus the array M of Fig. 1(b) has x_j as the first member of column j. If vertex x_i is the k^{th} nearest vertex to x_j, we have $m_{kj} = x_i$. As an example of this, it is clear that x_4 is the most distant

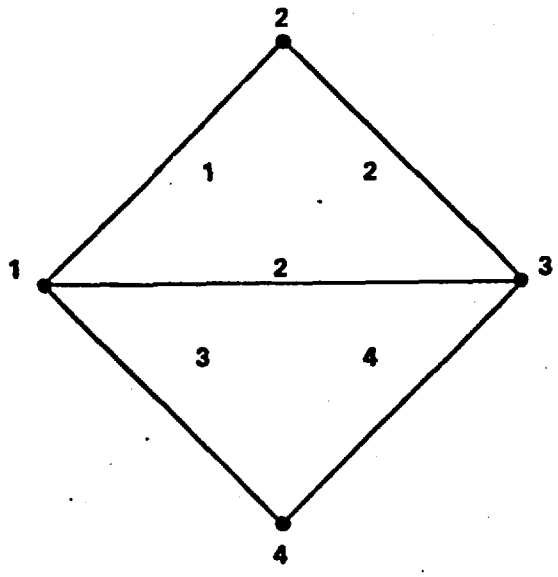

FIG 1(a). GRAPH FOR TREE SEARCH EXAMPLE

$$M = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_2 & x_1 & x_1 & x_1 \\ x_3 & x_3 & x_2 & x_2 \\ x_4 & x_4 & x_4 & x_3 \end{bmatrix}$$

FIG 1(b). M–ARRAY FOR GRAPH

$$\begin{bmatrix} x_1^{\dagger} & (x_2) & x_3 & x_4 \\ (x_2)_x & x_1^{\dagger} & x_1^{\dagger} & x_1 \\ x_3 & x_3 & x_2 & x_2 \\ x_4^{*} & x_4^{*} & x_4^{*} & x_3 \end{bmatrix}$$

FIG 1(c). M–ARRAY WITH MARKINGS – SEE TEXT

vertex from x_1 (3 units) and thus appears in the fourth row of the array. The search is by sequential allocation of all the vertices x_j of the graph: we can start with x_1 and finish with x_n - the graph can always be relabelled to permit this. Vertex x_j is allocated to m_{1j} $(=x_j)$, m_{2j}, ..., m_{nj}, in turn until all possibilities are implicitly enumerated.

We can make several observations about the nature of feasible solutions to the problem which are applicable to this search - or indeed any type of search using the same sort of information about the graph G.

(a) In the optimal solution there are p median vertices, thus each allocation of a vertex in this solution is the best of the p possible median vertex allocations. Hence, there are at least p-1 more costly ways of allocating any vertex. Therefore, the last p-1 rows of the array M may be permanently removed without any possibility of the optimal solution's being affected.

Unfortunately, this conclusion is not correct if the distance ordering of vertices is not unique, i.e. if two or more vertices are exactly the same distance from a third. For instance, in our example of Fig. 1, vertices x_3 and x_2 are both 4 units from x_4. This means that column 4 of M is not unique - so for a 2-median, that is, a situation in which we can usually omit the last row of M, we may only do this for the first three columns.

The same problem of non-uniqueness occurs in the remarks of the following paragraphs. We shall assume that by the use of the well-known degeneracy breaking trick† of small artificial costs, the matrix M can be made

† See any book on Linear Programming.

unique.

(b) Let us suppose that $m_{k'j'} = x_i$ and that vertex $x_{j'}$ has been allocated to a median vertex x_i. For column j of the M matrix, vertex x_i occurs in position k (say) - i.e. $m_{kj} = x_i$. Clearly all the entries $m_{\ell j}$ with $\ell > k$ can be neglected (marked), because the allocation of $x_{j'}$ to x_i implies that x_i is a median vertex, so that it is necessarily cheaper to use x_i for allocating x_j than any of the vertices $m_{\ell j}$ ($\ell > k$). This marking is <u>temporary</u> - if at some back tracking step in the search the allocation of $x_{j'}$ to x_i is negated, then the $m_{\ell j}$ entries must be freed for allocation, i.e. unmarked.

In our example, let us suppose that x_1 is a median vertex ($\xi_{11} = 1$). We may mark all the array M elements below the x_1 entry in each column. Thus in column 4 the x_2 and x_3 elements are marked, and similarly for the other columns.

(c) Suppose that vertex $x_{j'}$ is allocated to vertex $m_{k'j'}$ - this implies that all the vertices in column j' above this entry are not median vertices - for if they were, then an allocation to one of them would give lower resultant cost. Therefore, we may mark these vertices in all the columns which follow column j'. Thus, if in our example, vertex x_1 is allocated to vertex x_2, then clearly vertex x_1 is not a median vertex, so that it may be marked in all of the columns of the array. Again, it must be noted that this marking is purely temporary.

(d) A direct consequence of the statements about marking is that if the top t, say, of the entries in a column j are marked, and the next entry ($m_{t+1,j}$) is a

median vertex, then this vertex is the one to which x_j is allocated. No alternative need be considered until some of the top t entries are again unmarked as a result of a change in the previous allocations that produced some of the markings.

The tree search proceeds in the customary depth first fashion. Thus if at some stage p median vertices have been specified by the allocations already made, then any remaining unallocated vertices may now be allocated to their nearest median vertex neighbour. This is clearly the optimal completion of the partial solution as specified - the next step will be a backtracking step on the allocation made.

4.4.2 A lower bound for the algorithm

An implicit enumeration search of the type just described will not be very efficient unless the tree creation is truncated by methods which probe feasible solutions more deeply. The observations (a) - (d) made above, will limit the size of the tree search by reducing the number of alternative possible allocations of the vertices. As illustrated by the algorithms of the previous chapters, a lower bound on the value of the final optimal solution which could result from the allocations already made, can be used to limit the search further.

Suppose that at some stage of the search the last vertex to have been allocated is $x_{j'}$, and that p' vertices ($p' < p$) have been specified at this stage. Therefore, any further allocations must imply a total of an additional p-p' median vertices. Let J be the set of

indices of these as yet unallocated vertices - in general this will be the set of indices j where $j' < j \leq n$, excluding the indices of those vertices whose allocations have been forced by the allocations of the first j' vertices, as suggested by the remarks of the previous section. Let $m_{\alpha_j j}$ and $m_{\beta_j j}$ be the nearest and second nearest vertices to x_j still allowed, i.e. the first and second entries in column j which are unmarked. Clearly one would like to allocate x_j to $m_{\alpha_j j}$ from the point of view of cost. If the number of distinct vertices $m_{\alpha_j j}$ for $j \varepsilon J$ is k and $k = p - p'$, then all these best possible allocations for the, as yet, unallocated vertices are feasible in the sense of producing a total of p median vertices. Clearly these allocations give the optimal completion of the partial solution implied by the current allocations of the vertices $x_1, x_2, .., x_{j'}$. The result is noted and backtracking can occur from this partial solution.

In the case where $k > p-p'$ then at the very least $(k-p+p')$ of the best assignments must be changed to second best or worse in order to produce the required p vertices. Suppose vertex x_s is not used in the solution then an extra cost is incurred corresponding to each column j for which $x_s = m_{\alpha_j j}$, i.e. a total cost D_s where

$$D_s = \sum_{\substack{j \varepsilon J \\ x_s = m_{\alpha_j j}}} h_j \{ d(x_j, m_{\beta_j j}) - d(x_j, m_{\alpha_j j}) \}. \qquad (18)$$

For each vertex $x_s \varepsilon J$ there will be a cost D_s: a necessary consequence of finding a set of p vertices is that at least the $(k-p+p')$ <u>smallest</u> costs D_s are incurred in

addition to the necessary nearest neighbour costs. Hence a lower bound on the cost of an optimal solution derived by completion of the current partial solution is the sum of the (k-p+p') smallest D_s, as defined in equation (18), plus the cost of allocations already made for $x_1, x_2, .., x_j$, plus the sum of "best assignments", i.e.

$$\sum_{j \in J} h_j d(x_j, m_{\alpha_j j}).$$

As an illustration of the above, consider the graph of Fig. 1(a) again. Let us suppose that we are searching for the 2-median and that vertex x_1 has been allocated to median vertex x_2. Observation (a) means that the first three elements in the bottom row are permanently marked - we use a symbol "*" for this in Fig. 1(c). We have circled x_2 in column 1 to indicate that this is an allocation. Clearly observation (d) implies that x_2 is allocated to vertex x_2, so we circle this array entry also. From (c) we see that vertex x_1 is marked - we use † to indicate a temporary marking. In columns 3 and 4 the first entries are x_3 and x_4 respectively - the values D_3 and D_4 are seen to be 2 and 4. The value of p is 2, so that p' = 1 (we have found x_2 so far), while k, the number of distinct $m_{\alpha_j j}$ for $j \in J$ is 2 (x_3 and x_4) so that k-p+p' = 1, i.e. it is necessary that the smallest difference D_3 is incurred. Hence a lower bound on a completion of the present partial solution is: (1+0) (allocations) + (0+0) (minimum entries) + 2 (smallest difference) = 3.

4.5 An Alternative Direct Search Algorithm

The procedure of the previous section generates sub-problems by considering alternative allocations, i.e.

i.e. specifying values of the allocation ξ_{ij}. A different search procedure is to generate subproblems in a binary tree search by specifying whether or not a given vertex is to be a median vertex. This is a similar approach to that adopted in the problems of the two previous chapters. Thus the state of the search can be represented by sets S^+, S^- and F corresponding, respectively, to those vertices currently fixed as median vertices, those vertices fixed as non-median vertices and those vertices about which no decision has, as yet, been made.

The methods of the previous section enable us to calculate a lower bound on the cost of the optimal solution, given the decisions which have already been made about the vertices in S^+ and S^-. Hence, a vertex $x_j \varepsilon S^-$ must be allocated to a vertex in the set $S^+ \cup F$ - for the best allocation this will cost

$$u'_j = \underset{x_i \varepsilon S^+ \cup F}{\text{minimum}} \{h_j d(x_i, x_j)\}. \qquad (19)$$

Vertices in S^+ will, of course, cost nothing, as they will be allocated to themselves. A vertex $x_j \varepsilon F$ that is <u>not</u> going to become a median vertex will incur a cost of at least

$$u''_j = \underset{\substack{i \neq j \\ x_i \varepsilon S^+ \cup F}}{\text{minimum}} \{h_j d(x_i, x_j)\}. \qquad (20)$$

Now for a given partial solution, it is clear that not all of the free vertices can become median vertices in an optimal solution. In fact $n-p-|S^-|$ is the number of vertices which, although free at present, cannot become median vertices and hence must be reallocated to other vertices. Equation (20) gives the minimum cost which

would be incurred with a reallocation - thus a lower bound on any completion of the partial solution is

$$\sum_{x_j \in S^-} u'_j + U''$$

where U" is the sum of the $n-p-|S^-|$ smallest numbers u''_j. This is entirely analogous to the bound of the previous section.

The choice of the next vertex to be fixed open may be made in several ways. In the Appendix at the end of this chapter, we suggest one approach, in addition to the changes required if there are fixed costs.

Jarvinen et al (15) use a very similar bound to the above, in an isocost tree search, where branching always takes place from the tree vertex having the smallest lower bound. The main difference in the bound is that no vertices are fixed as median vertices. The use of such bounds, with modifications to deal with the generalised type of p-median problem encountered in practical depot location studies, can also be found in the literature (12,16).

4.6 Heuristic Methods with an Example

4.6.1 Some heuristic methods for the p-median

In this section we will briefly describe some heuristic methods which have been used to give 'approximate' sets of vertices for the p-median, i.e. sets of vertices with a small value for the total length L.

The method of Maranzana (17) has much in common with an iterative procedure for the continuous location problem due to Cooper (6). Essentially it is a partition-

ing method, in which the p-median is approached by finding the single vertex medians of p subsets of vertices each associated with one vertex, and then adjusting the subsets before repeating the process. There is no guarantee of optimality in this method which we will briefly describe here.

A subset V_o of p vertices is chosen from the vertex set X of the graph G. For each vertex $x_j \varepsilon V_o$ we find associated vertices P_{oj} such that

$$P_{oj} = \{x_i | d_{ij} \leq d_{ik}, \text{ all } x_k \varepsilon V_o\}. \tag{22}$$

Hence the collection $P_o = \{P_{oj} | x_j \varepsilon V_o\}$ of vertex subsets is a partition of X. Next the total weighted 'distance' (= 'cost') of the system for V_o as a p-median set is computed, i.e.

$$r_o = \sum_{x_j \varepsilon V_o} h_j \sum_{x_i \varepsilon P_{oj}} d_{ij} \tag{23}$$

For each P_{oj} we now find the single vertex median of the associated subgraph of G. This vertex median may be any vertex of P_{oj} including x_j - the set of medians now form a new subset V_1 of X. Now for each vertex $x_j \varepsilon V_1$ we can find a similarly associated set P_{1j} and compute r_1 - which may be shown to be less than r_o. We can see this by noting that the labelling of the vertex median for the partitioned sets P_{oj} as its new 'source' must either reduce or leave unchanged the weighted distance sum. Also, if a vertex is put into a new subset for the P_{1j}'s then its contribution to the total sum is reduced - hence no increase in the total distance sum can occur.

The process continues until after an iteration x_j is also the vertex median of the associated P_{tj} (for

the t^{th} iteration) and P_{tj} equals $P_{t-1,j}$: this being true for all the x_j. In this case the system has reached an equilibrium with the set V_t, an estimate of the p-median set.

Maranzana, while noting that there is no guarantee of optimality, claims that given several initial choices for V_o a solution very close in value to the p-median, or the actual p-median, will result. Generalisation to the situation with fixed costs is straightforward.

A heuristic method, based on vertex substitution, is described by Teitz and Bart (18), this method bearing a strong resemblance to a heuristic algorithm for the travelling salesman problem (19). The method proceeds by choosing any p vertices at random to form the initial set S which is assumed to be an approximation to the p-median set Y_p^*. The method then tests if any vertex $x_j \varepsilon X - S$ could replace a vertex $x_i \varepsilon S$ as a median vertex and so produce a new set $S' = S \cup \{x_j\} - x_i$ whose length L(S') is less than L(S). If so, the substitution of vertex x_i by x_j is performed, thus obtaining a set S' which is a better approximation to the p-median set Y_p^*. The same tests are now performed on the new set S' and so on, until a set $\bar{S}$ is finally obtained for which no substituion of a vertex in $\bar{S}$ by another vertex in $\bar{X} - \bar{S}$ produces a set with length less than $L(\bar{S})$. This final set $\bar{S}$ is then taken to be the required approximation to Y_p^*.

4.6.2 Description of the algorithm

Step 1. Select a set S of p vertices to form the initial approximation to the p-

median. Call all vertices $x_j \notin S$ "untried".

Step 2. Select some "untried" vertex $x_j \notin S$ and for each vertex $x_i \in S$, compute the "reduction" Δ_{ij} in the set length L(S) if x_j is substituted for x_i, i.e. compute:

$$\Delta_{ij} = L(S) - L(SU\{x_j\} - \{x_i\}) \qquad (24)$$

Step 3. Find $\Delta_{i_o j} = \max_{x_i \in S} (\Delta_{ij})$.

(i) If $\Delta_{i_o j} \leq 0$ call x_j "tried" and go to Step 2.

(ii) If $\Delta_{i_o j} > 0$ set $S \leftarrow SU\{x_j\} - \{x_i\}$ call x_j "tried" and go to Step 2.

Step 4. Repeat Steps 2 and 3 until all vertices in X-S have been tried. This is referred to as a cycle. If, during the last cycle no vertex substitution at all has been made at Step 3(i), go to Step 5. Otherwise, if some vertex substitution has been made, call all vertices "untried" and return to Step 2.

Step 5. Stop. The current set S is the estimated p-median set Y_p^*.

It is clear that this algorithm will not produce an optimal answer in all cases (15). An example of this is provided by the non-directed graph of Fig. 2, where link costs are shown next to the links and vertex weights are taken to be all unity. If the 2-median is required and the initial choice of the set S is $\{x_3, x_6\}$ with length L(S) = 8, then no substitution of a single vertex can lead to a set with lower length. However, the set $\{x_3, x_6\}$ is

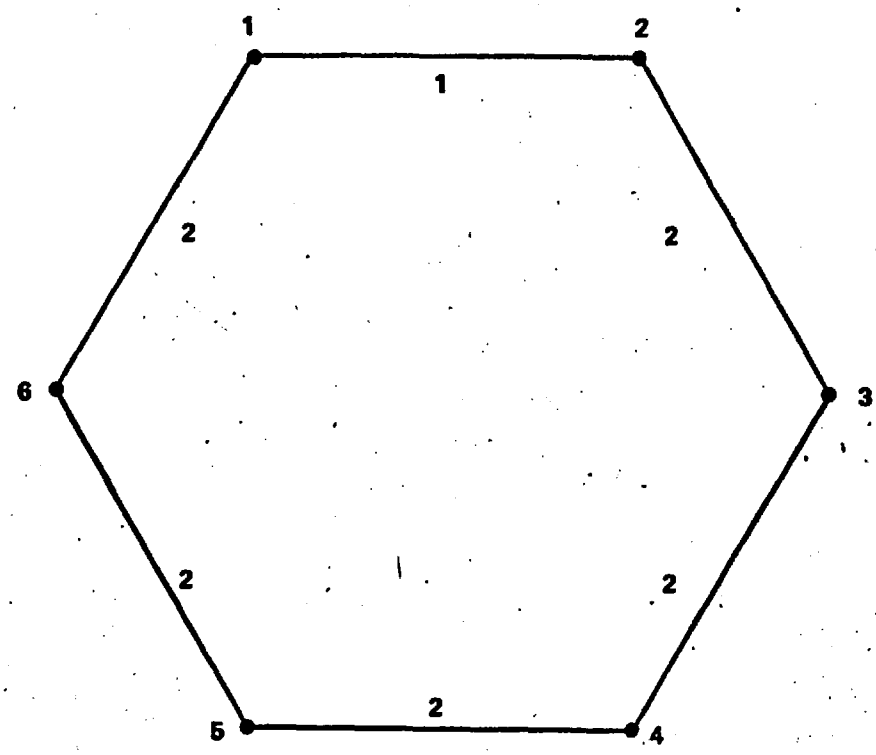

FIG 2. COUNTEREXAMPLE TO THE TEITZ AND BART ALGORITHM

not the 2-median of the graph and there are two 2-median sets $\{x_1, x_4\}$ and $\{x_2, x_5\}$ both with length $L(\bar{X}_2) = 7$.

The algorithm described in this section is in fact only one of a family of algorithms based on local optimisation and on the idea of ℓ-optimality first introduced by Lin (19) for the travelling salesman problem, subsequently extended and used by others (9,20) for a variety of combinatorial problems.

In the current problem, a set S of p vertices is called ℓ-optimal ($\ell \leq p$) - if the substitution of any ℓ vertices of S by other ℓ vertices not in S cannot produce a new set with length less than that of S. Obviously, if S is the p-median set Y_p^* of a graph, then S is p-optimal. It is also quite obvious that if S' is a ℓ'-optimal set and S" is a $\ell"$-optimal set, then, if $\ell" > \ell'$, $L(S") \leq L(S')$.

According to the above definitions, the answer produced by the algorithm of this section could be called 1-optimal, and similar algorithms producing 2-optimal, 3-optimal etc. answers could be described. In order to ensure that a given set S is ℓ-optimal, a total of:

$$\binom{p}{\ell} \times \binom{n-p}{\ell} \qquad (25)$$

potential substitutions (and hence calculations of lengths L) must be performed. This number increases rapidly with ℓ and hence practical algorithms cannot use values of ℓ much above 2 or 3.

REFERENCES

1. S.L. HAKIMI. "Optimum distribution of switching centres in a communications network and some related graph theoretical problems". Ops. Res., 13, pp. 462-475, 1965.

2. N. CHRISTOFIDES and P. VIOLA. "The optimum location of multicentres on a graph". Opl. Res. Quart., 22, pp. 145-154, 1971.

3. R.W. FLOYD. "Algorithm 97: shortest path". Comm. Ass. Com. Mach., 5, p. 345, 1962.

4. S.L. HAKIMI. "Optimum locations of switching centres and the absolute centres and medians of a graph". Ops. Res., 12, pp. 450-459, 1964.

5. A.J. GOLDMAN. "Optimal locations for centres in a network". Transp. Sci., 3, pp. 352-360, 1969.

6. L. COOPER. "Location-allocation problems". Ops. Res., 11, pp. 331-343, 1963.

7. G.L. CURRY and R.W. SKEITH. "A dynamic programming algorithm for facility location and allocation". AIIE Trans., 1, pp. 133-138, 1969.

8. L.B. ELLWEIN and P. GRAY. "Solving fixed charge location-allocation problems with capacity and configuration constraints". AIIE Trans., 3, pp. 290-298.

9. S. EILON, C.D.T. WATSON-GANDY and N. CHRISTOFIDES. "Distribution management: mathematical modelling and practical analysis". Griffin, London, 1971.

10. D.G. ELSON. "Site location via mixed-integer programming". Opl. Res. Quart., 23, pp. 31-44, 1972.

11. E. EFROYMSON and T.L. RAY. "A branch and bound algorithm for plant location". Ops. Res., 14, pp. 361-368, 1966.

12. K. SPIELBERG. "An algorithm for the simple plant location problem with some side conditions". Ops. Res., 17, pp. 85-111, 1969.

13. C.S. REVELLE and R.W. SWAIN. "Central facilities location". Geographical Analysis, 2, pp. 30-42, 1970.

14. R.E. MARSTON. "An algorithm for finding almost all the p-medians of a network". North Western University, Illinois, Discussion Paper 23, 1972.

15. P. JARVINEN, J. RAJALA and H. SINERVO. "A branch and bound algorithm for seeking the p-median". Ops. Res., 20, pp. 173-178, 1972.

16. K. SPIELBERG. "Plant location with generalised search origin". Man. Sci., 16, pp. 165-178, 1969.

17. F.E. MARANZANA. "On the location of supply points to minimise transport costs". Opl. Res. Quart., 15, pp. 261-270, 1964.

18. M.B. TEITZ and P. BART. "Heuristic methods for estimating the generalised vertex median of a weighted graph". Ops. Res., 16, pp. 995-961, 1968.

19. S. LIN. "Computer solutions of the travelling salesman problem". Bell Syst. Tech. Journal, 44, pp. 2245-2269, 1965.

20. B.W. KERNINGHAN and S. LIN. "An efficient heuristic procedure for partitioning graphs". Bell Syst. Tech. Journal, 49, pp. 296-307, 1970.

APPENDIX I

Fixed Costs and the p-Median Problem

When there are fixed costs the p-median problem is considerably altered. The existence of these costs enables extra possibilities for a tree search of the type we have described. Let us keep our notation of S^+, S^- and F for fixed open, fixed closed and free respectively - it helps to consider the problem from the point of view of depot location. We will refer to the weighted distance matrix as (d_{ij}) and the fixed costs as f_i.

Consider the calculation of a lower bound on a completion of a partial solution - to begin we rephrase the content of equation (18). For each column j of the cost matrix we find the smallest and second smallest costs - α_j and β_j respectively, the depot corresponding to α_j we call i_j. Thus if depot i is not used to supply customers (i.e. x_i is not a median vertex) an additional cost ρ_i is incurred, at the very least, compared with the situation when it is used, where

$$\rho_i = \sum_{\text{All } j \text{ s.t } i = i_j} (\beta_j - \alpha_j) \qquad \text{(A1)}$$

The costs ρ_i can be thought of as penalty costs on not using a depot. In the case where there are fixed costs it is apparent that a saving in cost of f_i will result if depot i is not used. Thus the total penalty cost for depot i is given by

$$\sigma_i = - f_i + \rho_i \qquad \text{(A2)}$$

For a given partial solution the fixed costs

corresponding to open depots must be incurred, while closed depots have been removed from consideration. We order the penalties σ_i for free depots in increasing order of magnitude, some of the σ_i may of course be negative. Now we can consider the ordered σ_i from the point of view of reducing the number of free depots in order to reach the best possible completion. There are $|S^+|$ depots fixed open and $|F|$ depots free - hence at least $(|F| - p + |S^+|)$ must be eliminated of the free depots. Thus the penalty costs σ_i for the $(|F| - p + |S^+|)$ smallest σ_i must be incurred. If there are any more <u>negative</u> σ_i's in the ascending sequence these too must be added on - in this case the implication is that it may not pay to have p vertices in the median set because of heavy fixed costs.

Thus the fixed costs can be dealt with quite simply, without much change to the original algorithm. Clearly it is important to know which of the alternative solutions, best set of p vertices, or best set of k vertices ($k \leq p$), is desired.

Fixed costs are also important in deciding which depot to keep open in the tree search - we give a simple example of how the direction of the search can be decided upon. It would seem sensible to open depots where the benefit in variable cost reduction outweighs the added fixed cost. Conversely, an even stronger statement, a depot which does 'pay for itself' in this way must be fixed closed - the variable cost benefit being calculated with respect to open depots only.

Suppose x_i is a free depot, then define

$$w_{ij} = \underset{x_k \in S^+}{\text{Minimum}} \ (\text{Maximum} \ (d_{kj} - d_{ij}, 0)), \qquad \text{(A3)}$$

$$\phi_i = \sum_{\text{all } j} w_{ij} - f_i \qquad \text{(A4)}$$

If $\phi_i < 0$ then depot i should be fixed closed, while the depot with the largest value of ϕ_i is a good candidate for fixing open. In the cases where there are no fixed open depots we can heuristically choose the depot with the largest penalty cost - or the lowest value of this times the number of customers for which the depot is nearest.

For zero fixed costs it is still possible to use this test to decide upon which depot to fix open.

APPENDIX II

In Section 4.5 we discussed a Direct Search Algorithm for the p-median problem. In this Appendix, we give some idea of the computing efficiency of this method. The criterion of efficiency we adopt is the central processing time (cp time) taken by the algorithm for randomly generated problems. Table A1 shows the computer cp times for a selection of randomly generated graphs (both directed and undirected links in all the graphs - average vertex in-degree of about 3). The number of vertices (N) is 10, 20 and 30, while the value of p ranges from 2 to 28. The cp times should be compared with those resulting from other methods - see Khumawala et al. (Management Science vol. 21, no. 2, October 1974, pp. 230-234) for example.

TABLE A1

COMPUTING TIMES FOR A DIRECT SEARCH ALGORITHM FOR THE p-MEDIAN PROBLEM

	CPU TIMES: CDC 6400 SECONDS		
	10	20	30
2	0.117(3,9)	2.48(3,85)	5.44(3,39)
4	0.165(11,15)	1.67(5,67)	40.21(33,669)
6	0.137(7,13)	1.01(7,49)	41.05(7,919)
8	0.159(9,17)	1.24(9,63)	24.03(9,673)
10		0.76(11,35)	15.65(69,513)
12		1.34(75,87)	16.70(223,681)
14		1.85(69,83)	9.99(217,425)
16		0.52(17,33)	8.45(125,397)
18		0.53(19,37)	3.32(63,123)
20			3.67(75,163)
22			3.31(23,131)
24			1.82(25,49)
26			1.88(27,53)
28			1.87(29,57)

The bracketed figures are the number of bound calculations to the optimum solution and to the end of the search.