STAN-CS-72-293

SEL-72-027

PB 212 234

Combinatorial Solutions to Partitioning Problems

by

J. A. Lukes

May 1972

Technical Report No. 32

This work has been supported by the National Science Foundation under Grant GJ-1180 and by Dr. Lukes' fellowship from the IBM Corporation.

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY • STANFORD, CALIFORNIA

BIBLIOGRAPHIC DATA SHEET	1. Report No. STAN-CS-72-293	2.	3. Recipient's Accession No. PB-212 234
4. Title and Subtitle Combinatorial Solutions to Partitioning Problems			5. Report Date May 1972 6.
7. Author(s) J. A. Lukes			8. Performing Organization Rept. No. STAN-CS-72-293
9. Performing Organization Name and Address Stanford University Computer Science Department Stanford, California 94305			10. Project/Task/Work Unit No. 11. Contract/Grant No. GJ1180
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C.			13. Type of Report & Period Covered technical 14.

15. Supplementary Notes

16. Abstracts

In this dissertation we describe algorithms that use graph properties and dynamic programming techniques to generate the optimal partition of an arbitrary graph. In particular, let G be a graph with weighted nodes and weighted edges. We consider algorithms that solve the problem of partitioning G into clusters of nodes such that the sum of the node weights in any cluster does not exceed a given maximum W and the weights of the intercluster edges are minimized. An interesting application of such an algorithm is the assignment of a program's subroutines and data to pages in a paged memory system so as to minimize paging faults.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Distribution unlimited 1a	UNCLASSIFIED	124
	20. Security Class (This Page) UNCLASSIFIED	22. Price $3.00

STAN-CS-72-293

SEL-72-027

COMBINATORIAL SOLUTIONS TO PARTITIONING PROBLEMS

by

J. A. Lukes

May 1972

Technical Report no. 32

DIGITAL SYSTEMS LABORATORY

Department of Electrical Engineering

Department of Computer Science

Stanford University

Stanford, California

1b

This work was supported by the National Science Foundation under grant GJ-1180 and by Dr. Lukes' fellowship from the IBM Corporation.

ABSTRACT

In this dissertation we describe algorithms that use graph properties and dynamic programming techniques to generate the optimal partition of an arbitrary graph. In particular, let G be a graph with weighted nodes and weighted edges. We consider algorithms that solve the problem of partitioning G into clusters of nodes such that the sum of the node weights in any cluster does not exceed a given maximum W and the weights of the inter-cluster edges are minimized. An interesting application of such an algorithm is the assignment of a program's subroutines and data to pages in a paged memory system so as to minimize paging faults.

The concepts of dynamic programming and, in particular, those techniques appropriate to the solution of the "knapsack" problem, are employed in an algorithm that generates the optimal partition of an arbitrary graph. An upper bound on the algorithm's growth in computation time and storage to partition an n node graph is

$$n^4\left\lceil\frac{nW}{e}\right\rceil^n$$

where e is the base of the natural logarithms. We use the following graph properties to reduce this growth rate:

(1) the degree of the graph;

(2) the existence of cutpoints in the graph.

The first property bounds the growth in time and storage of the algorithm to less than

$$x^2n^2\left\lceil\frac{xW}{e}\right\rceil^x$$

where x is a function of the degree of the graph. The value of x is independent of the number of nodes in the graph; however, the degree of a graph may grow as n. A graph whose nodes are adjacent to few others has a value of $x \ll n$ and, for a small value of W, can be partitioned very efficiently.

If any n node graph G contains one or more cutpoints, we show that G can be partitioned by partitioning the blocks of G and combining these partitions. A considerable reduction in time and storage to partition the graph results if the number of nodes in each block of G is small compared to n.

A very efficient variation of the general algorithm results if the graph to be partitioned is a tree. We show that trees can be partitioned in a time proportional to the number of nodes in the graph.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I wish to thank my major advisor, Professor H. S. Stone, for his excellent advice, undying support, and phenomenal patience during the writing of this dissertation. Professors A. M. Peterson and R. W. Dutton are also to be thanked for their excellent suggestions in presenting the work reported here.

I also wish to thank International Business Machines Corporation for its financial assistance in the form of a Resident Study Fellowship and my wife for her encouragement and patience.

CHAPTER I

INTRODUCTION

Consider a graph G whose nodes have nonnegative integer weights and whose edges have positive values. A familiar combinatorial problem is the partitioning of G into subgraphs such that the sum of the node weights in any subgraph does not exceed a given maximum and the sum of the values of the edges joining different subgraphs is minimal.

An interesting example of this partitioning problem is that of partitioning a program to be run on a computer with a paged memory system into pages so that paging faults are minimized [Kernighan, 1971]. Here the graph is the program and the nodes are collections of instructions (such as subroutines) or data (such as arrays) making up that program. The edges are the transitions that might occur from one subroutine to other subroutines and data, for example.

Before describing previous investigations of partitioning problems, we define partition in the sense used here.

A. PROBLEM DEFINITION AND RESTRICTIONS

Given a graph $G=(V,E)$ with node set V and edge set E, a partition of G is a collection of k clusters of nodes $\{c_i\}$ $(i=1,2,\ldots,k)$ such that

(1) $\bigcup_{i=1}^{k} c_i = V$,

(2) $c_i \cap c_j = \emptyset$ for all $i \neq j$.

A nonnegative integer weight w_i is associated with each node i and a positive value v_{ij} with each edge (i,j). A weight constraint is imposed upon each cluster of a partition. Given a positive, integer weight constraint W, the

sum of the node weights in any cluster of a partition must not exceed W. An edge (a,b) is cut by a partition if nodes a and b are in different clusters. Fig. 1.1 illustrates a partition of the given graph where the weight constraint is two.

An optimal partition is defined as some partition of G, $p_G(opt) = \{c_1, c_2, \ldots, c_k\}$, with the property that each cluster c_i satisfies the weight constraint,

$$\sum_{j \in c_i} w_j \leq W,$$

and

$$\sum_{\substack{i \in c_f \\ \text{and} \\ j \in c_g}} v_{ij} \quad \text{is minimal} \quad (f,g=1,2,\ldots,k). \quad f \neq g$$

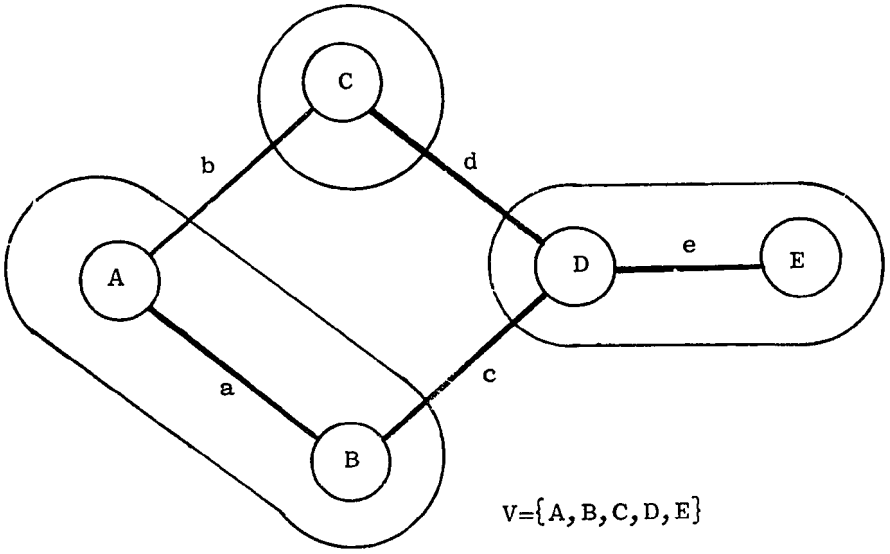
An equivalent property is that each cluster satisfies the weight constraint and

$$\sum_{i,j \in c_f} v_{ij} \quad \text{is maximal} \quad (f=1,2,\ldots,k),$$

since the sum of the edges in clusters plus the sum of the edges cut by the partition equals the sum of the values of G's edges.

We impose several restrictions on the problem investigated here. The first is that the nodes of the graph must have nonnegative integer weights. The only restriction placed upon the values of the edges, however, is that they are positive. Another restriction is that the graph be connected*. Given a disconnected graph G, each connected subgraph of G is partitioned

* A connected graph has a path from any node in the graph to all other nodes in the graph.



$V=\{A,B,C,D,E\}$

$E=\{a,b,c,d,e\}$

weight constraint = 2

all nodes have unit weight

$$\text{partition} = \{c_1, c_2, c_3\}$$

$$c_1 = (A,B)$$

$$c_2 = (C)$$

$$c_3 = (D,E)$$

Figure 1.1 -- A partition of the graph G=(V,E)

independently -- i.e., each cluster consists of nodes from the same connected subgraph. This restriction does not affect the optimality of the solution to the partitioning problem -- a proof of this fact is given in Chapter II.

The final restriction is that a multigraph must be transformed into a graph by the following modification. If more than one edge exists between two nodes i and j, then the several edges joining i and j are replaced by one with a value v_{ij} equal to the sum of the values of those edges.

B. HISTORY

The partitioning problem defined above is one of several found in the literature on optimal partitioning. Two others frequently investigated are the following:

(1) Partition a graph with weighted nodes into clusters so that each cluster does not exceed a given weight constraint and the number of clusters is a minimum.

(2) Partition a directed graph with weighted nodes and edges with values that are zero if the edge is in a cluster and positive if cut by the partition. The objective in this problem is to minimize the value of the worst-case directed path by clustering the given network under both weight and pin constraints*.

An example of the first problem is that of packaging a logic design with the objective of minimizing the number of clusters required. The second problem occurs in the packaging of a logic design when the objective is to minimize the delay associated with intercluster wiring.

* A pin constraint is a restriction placed on the number of edges cut by each cluster of a partition.

The literature on optimal partitioning generally falls into the three categories above. Lawler [1962], Luccio and Sami [1968], and Kernighan [1971] have investigated restricted problems of the type considered here. Stone [1970] has investigated the problem of minimizing the number of modules required to partition a logic network. Lawler, Levitt, and Turner [1969] and Jensen [1970] have investigated the problem of partitioning a directed acyclic graph with the objective of minimizing the maximum-delay path.

C. A COMBINATORIAL APPROACH TO THE PARTITIONING PROBLEM

In this thesis we describe combinatorial algorithms that use graph properties and a dynamic programming procedure to generate the optimal partition of a connected graph.

The dynamic programming procedure generates "feasible" partitions, i.e. those partitions of a graph G whose clusters satisfy the weight constraint and form connected subgraphs of G. The number of feasible partitions of a k node graph grows exponentially in k, consequently we use certain graph properties to reduce the number of feasible partitions generated on each stage of the dynamic programming procedure. These properties are:

(1) the number of nodes adjacent to each node of the graph;

(2) the existence of cutpoints in the graph.

The first property limits the number of partitions generated on the kth stage of the dynamic programming procedure, p_k, to

$$p_k \leq x_k \left[\frac{x_k W}{e}\right]^{x_k},$$

where x_k is a function of the degree of the graph, e is the base of the natural logarithms, and W is the weight constraint. Note that x_k is independent of the number of nodes in the graph.

The growth in computation time for the kth step of the partitioning algorithm is proportional to

$$n[p_k(\log_2 p_k)]$$

where n is the number of graph nodes. The computation time is proportional to

$$n \sum_{k=1}^{n} p_k \log_2 p_k$$

therefore it grows asymptotically as

$$n^2 \; p \log_2 p \doteq n^2 x^2 \left(\frac{xW}{e}\right)^x$$

where

$$p = x\left[\frac{xW}{e}\right]^x$$

and

$$x = \max_{1 \le k \le n} \{x_k\} .$$

The storage requirements grow asymptotically as

$$np=nx\left(\frac{xW}{e}\right)^x .$$

We show that an algorithm that generates all feasible partitions of an n node graph G has an asymptotic growth in computation time of

$$n^2 \; p \log_2 p \doteq n^4 \left(\frac{nW}{e}\right)^n$$

where

$$p=(n)\left(\frac{nW}{e}\right)^n .$$

A comparison of the partitioning algorithm developed in this thesis and an algorithm that generates all feasible partitions shows a reduction in the growth in computation time and storage of

$$n^{n-x} .$$

If each node of G is adjacent to few others, the value of x is much less than n resulting in a significant reduction in computation time over a procedure that simply generates all feasible partitions.

A graph with cutpoints can be partitioned by first partitioning the blocks of the graph, then combining these partitions to form the optimal partition of the entire graph. The maximum number of partitions generated on a step of the partitioning process is a function of the number of nodes in a block of the graph, not the graph itself.

The special properties of a graph in the form of a tree are used to create an algorithm for tree-partitioning. This algorithm has a growth in computation time and storage requirements that varies linearly with the number of graph nodes.

CHAPTER II

A GENERAL GRAPH PARTITIONING ALGORITHM

In this chapter we describe a partitioning algorithm that has as its basis a dynamic programming procedure similar to that used in the solution of the one-dimensional knapsack problem [Gilmore, Gomory, 1966]. The similarity between that problem and the partitioning problem becomes apparent when their properties are compared.

The one-dimensional knapsack problem can be posed as the problem faced by a mountain climber who has a knapsack that can carry a maximum weight of W pounds and a number of different items he wishes to carry in the knapsack. Each item has a weight and value associated with it, and the sum of the weights of the items exceeds W. A mathematical statement of this problem is the following:

1-dimensional knapsack problem

Let w_i = weight of item i
v_i = value of item i
W = capacity of knapsack
$(i=1,2,\ldots,n)$

Maximize $\sum_{i=1}^{n} v_i x_i$ subject to the constraints

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i = \begin{cases} 1 & \text{if item i is in knapsack} \\ 0 & \text{otherwise.} \end{cases}$$

The mathematical statement of the partitioning problem, given below, is seen to be an extension of the one-dimensional knapsack problem to the distribution of interconnected, weighted items into many "knapsacks" or clusters, each of capacity W:

Partitioning problem

Let w_i = weight of node i

v_{ij} = value of edge (i,j) $\quad (i,j=1,2,\ldots,n)$

W = weight constraint

Maximize $\sum_{i=1}^{n} \sum_{j=1}^{n} v_{ij} x_{ik} x_{jk}$ subject to the constraints

$$\sum_{i=1}^{n} w_i x_{ik} \leq W \qquad (k=1,2,\ldots,\text{number of clusters in partition})$$

$$x_{ik} = \begin{cases} 1 & \text{if node i is in cluster k} \\ 0 & \text{otherwise.} \end{cases}$$

A problem amenable to solution using dynamic programming must have the following characteristics [Hillier and Lieberman, 1967]:

(1) The problem is divisible into stages with a policy decision required at each stage.

(2) Each stage has a number of states associated with it.

(3) The policy decision translates a state associated with the present stage into a state associated with the next stage.

(4) Given the current state, an optimal policy for the remaining stages is independent of how the current state is reached.

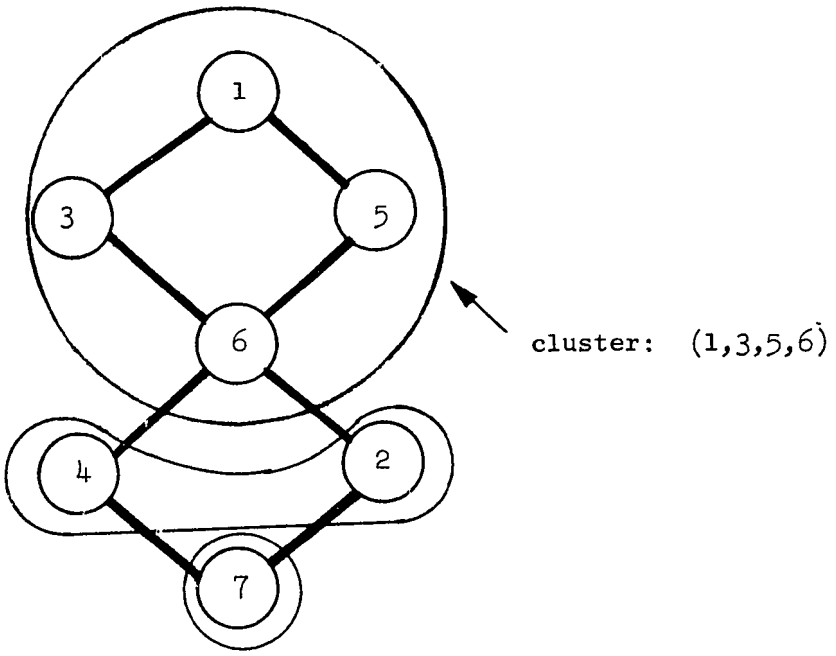
We now show that the partitioning problem satisfies these characteristics.

In order to pose the partitioning problem as one suitable for solution by dynamic programming, the graph is first labeled. A labeling is the assignment of a unique integer to each node of the graph; the node associated with some integer k by the labeling is then referred to as "node k." The kth step, or stage, of the partitioning process generates the feasible partitions of the subgraph consisting of those nodes with

labels $\leq$ k. These partitions correspond to the states of the kth stage. The partitions of the subgraph consisting of those nodes with labels no greater than k are created from the partitions of the k-1 st step by adding node k to these partitions within the limitations imposed by the weight constraint. The policy decision is the determination of which partitions of step k-1 can have node k added to one of their clusters to generate partitions of step k. Consequently, it is apparent that the partitioning problem can be solved with a dynamic programming procedure. Before describing the basic partitioning process, we give the following definitions.

A. DEFINITIONS

The nodal representation of a partition is an unordered collection of lists where each list represents a cluster and the contents of the list are the nodes in that cluster. For example, a cluster with nodes 1,3,5, and 6 is represented by the list (1,3,5,6), where the order in which the nodes appear in the list is not important. An example of a nodal representation of a partition with this cluster is (1,3,5,6)(2,4)(7). The set of partitions generated on the kth step of the partitioning process are denoted by P_k. A partition in P_k is denoted by $p_{i,k}$ and represents a partition of the subgraph consisting of nodes with labels less than or equal to k. The value of some partition $p_{i,k}$ is defined as the sum of the values of the edges within the clusters of $p_{i,k}$. The weight of a cluster is defined as the sum of the weights of the nodes in that cluster. The cost of a partition equals the sum of the values of the intercluster edges. The cost plus the value of a partition equals the sum of the values of all edges in the graph for which that partition is generated. Fig. 2.1 illustrates several of these definitions.

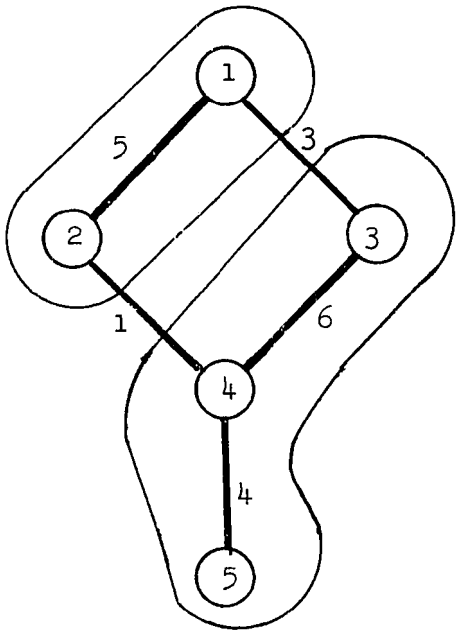


partition: (1,3,5,6)(2,4)(7)

(a) Nodal representation of a partition

Figure 2.1 -- Examples of definitions

partition p:

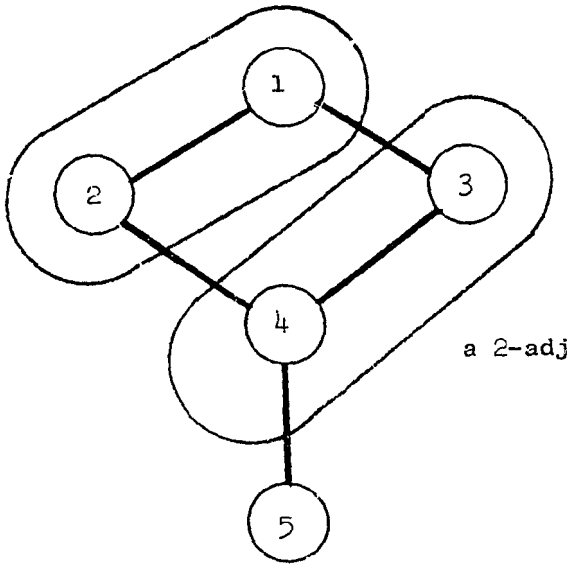


VALUE(p) = 6 + 4 + 5 = 15

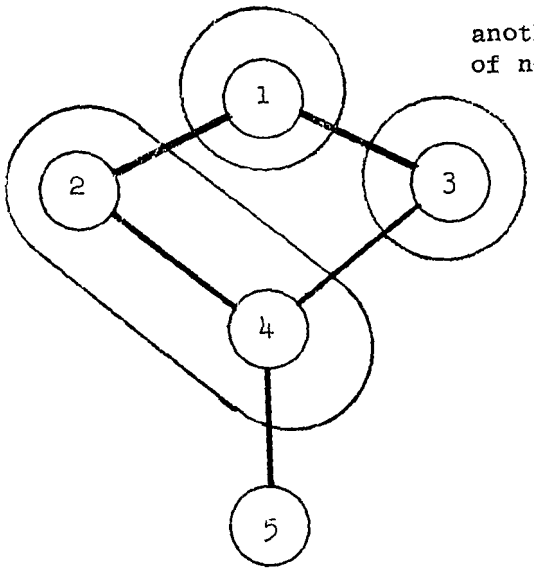
COST(p) = 3 + 1 = 4

(b) Value and cost of a partition

Figure 2.1 -- Examples of definitions



a 2-adjacency of node 4:
(1,2)(3,4)



another 2-adjacency
of node 4:
(1)(3)(2,4)

(c) A k-adjacency of node j

Figure 2.1 -- Examples of definitions

A k-adjacency of node j is defined as a partition $p_{i,j}$ with a cluster containing node j whose weight is k. An example of a 2-adjacency of node 4 is shown in Fig. 2.1. A k-adjacency of a node is not unique, as is also shown in Fig. 2.1.

In Chapter I a weight constraint is imposed upon each cluster of a partition. The following theorem further constrains the properties of the nodes in a cluster.

Connectivity Theorem

An optimal policy for a connected graph G is to cluster only those nodes that ultimately form a connected subgraph of G.

Proof

Let there exist a cluster of an optimal partition of G that contains two or more disjoint connected subgraphs, $S_1, S_2, \ldots, S_k$. Since the graph G is connected, some subgraph S_i can be removed from the cluster in which it presently exists and added to a cluster in which there is at least one node adjacent to some node of S_i. If the sum of the nodes in the newly formed cluster does not exceed the weight constraint, an edge that was cut by the partition (there may be more than one) is now within the newly formed cluster. Since all edge values are positive, the original partition is not optimal, contrary to the given condition; this contradiction proves the theorem.

If the subgraph S_i cannot be added to a cluster containing a node adjacent to some node of S_i without violating the weight limitation, then it can be clustered by itself with no increase in the cost of the partition. Consequently, all clusters in an optimal partition of a connected graph G can form connected subgraphs of G. A partition of G can, however, have clusters containing disjoint connected subgraphs with a cost that is equal

to that of the optimal partition generated using this policy, and, in fact, may require fewer clusters. The point of the theorem is that the connectivity limitation does not cause the deletion of an optimal partition. ■

A feasible partition of a graph G is defined as a partition whose clusters each satisfy the following properties:

(1) The sum of the weights of the nodes in a cluster must not exceed W, the weight constraint.

(2) The nodes in a cluster must form a connected subgraph of G.

In the process of partitioning a connected graph G the only partitions that need to be generated are those whose clusters have a weight not exceeding the weight constraint and that contain nodes that may form a connected subgraph of G. In generating the set of partitions P_k on step k, the weight constraint is easily tested by adding node k to each cluster of some partition in P_{k-1} and rejecting the resulting partitions with a cluster whose weight exceeds W. A newly created element of P_k must not only have clusters that satisfy the weight constraint, but its clusters must also contain nodes that presently form a connected subgraph, or form a connected subgraph with the addition of one or more nodes with labels greater than k. Let this restriction be called the connectivity constraint. In order to recognize some cluster of an element of P_{k-1} to which node k can be added without violating the connectivity constraint, we introduce the concept of the connected set.

The connected set for a node k is defined as that set of nodes that, if one or more of them appears in a cluster of a partition in P_{k-1}, guarantees that the addition of node k to that cluster may on some step $j \geq k$ form a connected subgraph. The properties of a node i in the connected set for node k, denoted by CONN(k), are:

(1) $i < k$;

(2) node i

(a) is adjacent to node k, or

(b) lies on a path $i,j_1,j_2,\ldots,j_r,k$ where

$$\sum_{y \in \{i,j_1,\ldots,k\}} \text{WEIGHT}[y] \leq W$$

$$j_m > k \quad \text{for } m=1,2,\ldots,r.$$

The second property guarantees that a partition with a cluster containing two nodes i and k that are presently disconnected, but become connected if nodes $j_1,j_2,\ldots,j_r$ are added to that cluster, is generated on step k.

An illustration of the connected set associated with each node of the given graph is shown in Fig. 2.2.

B. DYNAMIC PROGRAMMING PROCEDURE

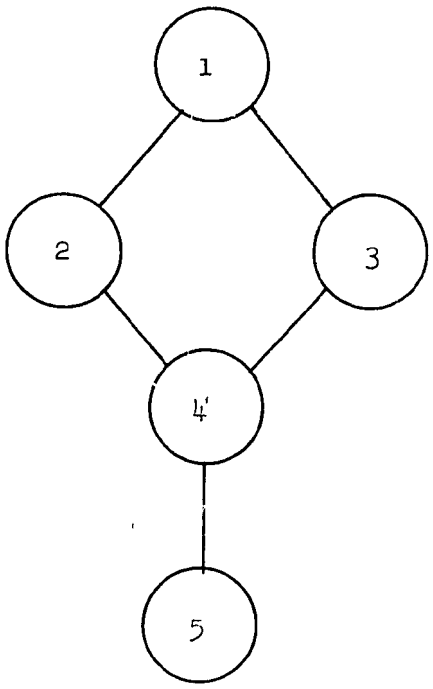
We now describe the dynamic programming procedure that forms the basis of the partitioning algorithm. A labeling is assumed to have been impressed upon the graph. The particular labeling used affects the partitioning process. Chapter IV discusses the problem of labeling a graph.

The kth step of the partitioning algorithm has as its states the partitions of the subgraph consisting of those nodes with labels $\leq k$, denoted by P_k. We then add node k+1 to all partitions in P_k with a cluster satisfying the criteria:

(1) the addition of node k+1 does not cause the cluster weight to exceed the weight constraint;

(2) there exists a node in CONN(k+1), the connected set for node k+1, in the cluster.

The resulting partitions are the states of k+1, P_{k+1}.



W = 3

all nodes
unit weight

CONN(1) = ϕ

CONN(2) = {1}

CONN(3) = {1,2}

CONN(4) = {2,3}

CONN(5) = {4}

Figure 2.2 -- Illustration of connected set

The value of each partition equals the summation of the edges within clusters of the partition. This is expressed as follows:

$$\text{VALUE}[p_{x,k+1}] = \sum_{y} \text{VALUE}[\text{edge}(y,k+1)] + \text{VALUE}[p_{z,k}]$$

where

$\text{VALUE}[p_{z,k}]$ = value of a partition in P_k--$p_{z,k}$

$y \in \text{CONN}(k+1)'$ where $\text{CONN}(k+1)'$ is the subset of $\text{CONN}(k+1)$ present in the cluster of $p_{z,k}$ to which node k+1 was added.

The dynamic programming process is outlined below:

STEP 1

For each node k find the connected set, CONN(k).

STEP 2

$j=0$, $P_0=\phi$

STEP 3

$j=j+1$

Let the weight of node j be denoted by w_j. P_j consists of the following partitions:

(a) Form the k-adjacencies for $k=w_j$.

Each such k-adjacency is formed by adding a cluster containing node j alone to the set of clusters of a partition in P_{j-1}.

(b) For $k=w_j+1,\ldots,W$, form the k-adjacencies of node j. Only those partitions in P_{j-1} with at least one cluster containing a node in CONN(j) can generate these partitions.

STEP 4

Go to step 3 until j=n for an n node graph.

STEP 5

Select the maximal-valued partition in P_n. This is the optimal-valued partition of the graph.

We prove the optimality of the dynamic programming procedure by the following argument. The Connectivity Theorem shows that no partition of an n node graph G can have a value greater than a partition each of whose clusters forms a connected subgraph of G. We must then show that the above algorithm generates all such partitions.

On step k of the algorithm node k is added to the clusters in each partition in P_{k-1} such that neither the weight nor the connectivity constraint is violated. The algorithm may, however, fail to generate an optimal partition of G if on some step k the addition of node k to a cluster violating either the weight or the connectivity constraint results in a feasible partition of G.

If node k is added to a cluster of some partition in P_{k-1} and the resulting partition $p_{j,k}$ contains a cluster that violates the weight constraint, it is clear that all partitions derived from $p_{j,k}$ also have a cluster that violates the weight constraint. This result follows from the fact that a node is never removed from a cluster on some step of the algorithm, and each node has a nonnegative weight.

Let node k be added to a (nonvoid) cluster c of a partition in P_{k-1}. If the set of nodes in c is $\{i_1,i_2,\ldots,i_r\}$, then the addition of node k to c violates the connectivity constraint if:

(1) no node in c is adjacent to k;

(2) given any node i_m in c, there is no path $i_m,j_1,\ldots,j_z,k$ such that

$$\sum_{y} \text{WEIGHT}[y] \leq W$$

where

$$j_h > k \quad \text{for } h=1,2,\ldots,z$$

$$y \in \{i_m,j_1,\ldots,j_z,k\} .$$

Let the partition in P_k generated by adding node k to cluster c be denoted by $p_{j,k}$. Then, every partition of G derived from $p_{j,k}$ has a cluster containing nodes $k,i_1,i_2,\ldots,i_r$ in which there is no path (within the cluster) from node k to any of the nodes $i_1,i_2,\ldots,i_r$. As a consequence, that cluster cannot form a connected subgraph of G.

In conclusion, the addition of some node k to a cluster violating either the weight or the connectivity constraint cannot result in a partition of G whose clusters satisfy the weight constraint and form connected subgraphs of G. The algorithm described above therefore generates the optimal partition of graph G.

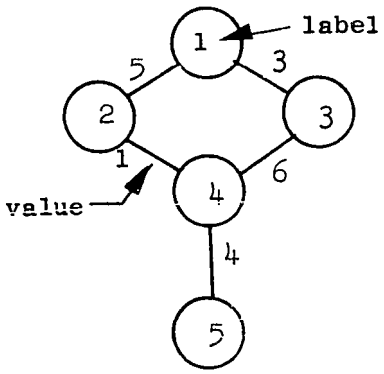
An example of the use of this algorithm is given in Fig. 2.3. The results of each step of the algorithm are contained in a tabular form. Each row of this table corresponds to a step of the procedure; the kth column and jth row of the table contain the k-adjacencies of node j.

C. GROWTH RATE FOR DYNAMIC PROGRAMMING PROCEDURE

Although the dynamic programming procedure just described generates an optimal partition of a graph without resorting to total enumeration, the question arises as to the number of feasible partitions possible for a connected graph. In Appendix B we show that the growth in computation time varies as

$$\sum_{k=1}^{n} np_k[\log_2 p_k]$$

and the storage requirements vary as np_k, where n equals the number of graph nodes and p_k the number of partitions in the set P_k generated on the kth step of the dynamic programming procedure.



all nodes unit weight

W = 3

NODE	CONNECTED SET
1	ϕ
2	{1}
3	{1,2}
4	{2,3}
5	{4}

k-adjacencies

STEP	1	2	3
1	(1) = 0		
2	(1)(2) = 0	(1,2) = 5	
3	(1)(2)(3) = 0 (1,2)(3) = 5	(1,3)(2) = 3 (1)(2,3) = 0	(1,2,3) = 8
4	(1)(2)(3)(4) = 0 (1,2)(3)(4) = 5 (1,3)(2)(4) = 3 (1,2,3)(4) = 8	(1)(2,4)(3) = 1 (1)(2)(3,4) = 6 (1,3)(2,4) = 4 (1,2)(3,4) = 11	(1,2,4)(3) = 6 (1,3,4)(2) = 9 (1)(2,3,4) = 7
5	(1)(2)(3)(4)(5) = 0 (1,2)(3)(4)(5) = 5 (1,3)(2)(4)(5) = 3 (1,2,3)(4)(5) = 8 (1)(2,4)(3)(5) = 1 (1)(2)(3,4)(5) = 6 (1,3)(2,4)(5) = 4 (1,2)(3,4)(5) = 11 (1,2,4)(3)(5) = 6 (1,3,4)(2)(5) = 9 (1)(2,3,4)(5) = 7	(1)(2)(3)(4,5) = 4 (1,2)(3)(4,5) = 9 (1,3)(2)(4,5) = 7 (1,2,3)(4,5) = 12 thus optimal partition is (1,2)(3,4,5) VALUE = 15	(1)(2,4,5)(3) = 5 (1)(2)(3,4,5) = 10 (1,3)(2,4,5) = 8 (1,2)(3,4,5) = 15

Figure 2.3 -- Example of the dynamic programming procedure

Consider first the growth in the cardinality of P_k for total enumeration. To generate this number we assume that the graph is complete* so that no combination of nodes in some cluster is disconnected. Also, no weight constraint is imposed upon the clusters. The upper bound on the size of P_k,p_k, is the number of ways in which k distinct objects can be distributed in i nondistinct cells, where i varies from one to k. The Stirling number of the second kind, S(k,i), enumerates the ways in which k distinct objects can be distributed into i nondistinct cells, where no cell is left empty. Thus

$$p_k < \sum_{i=1}^{k} S(k,i) \ .$$

A closed form for this summation does not appear to exist, but an upper bound results from the recurrence relationship:

$$p_k < (1+c_k)\ p_{k-1} \qquad \text{for } p_1=1 \text{ and where } c_k=\left|\overline{\text{CONN}(k)}\right| \ .$$

This relationship is derived from the fact that P_k is made up of two subsets:

(1) the 1-adjacencies of P_k, of which there are p_{k-1};

(2) the k-adjacencies of P_k, where $k > 1$.

The size of the latter set is bounded by $c_k p_{k-1}$ since each node in CONN(k) can generate no more than p_{k-1} partitions of P_k.

For the complete graph, $\left|\text{CONN}(k)\right| = k-1$, therefore

$$p_k < k! \ .$$

* A complete graph has every pair of its nodes adjacent.

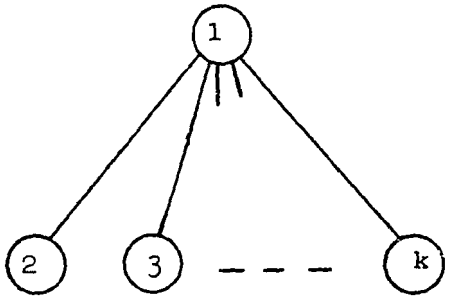
Consequently, an upper bound on the number of partitions generated on the kth step of the partitioning process is k!.

To derive a lower bound on the number of feasible partitions, consider the two trees of Fig. 2.4. The size of CONN(k) for k > 1 is one for both of these trees. The tree of Fig. 2.4(a) has the property that CONN(k) is the same for all k, whereas that of Fig. 2.4(b) has no two CONN(k) equal.

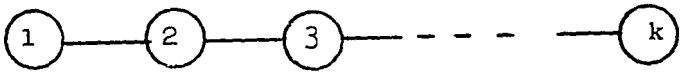
Since every connected graph has a spanning tree* [Liu, 1968], there is at least one labeling of G such that $|\text{CONN}(k)| \geq 1$ for each node k > 1. This result follows from the fact that a spanning tree can always be labeled so that the label of a branch node is less than those of its sons. The number of feasible partitions generated on the kth step of the dynamic programming procedure increases with the size of CONN(k). Also, the number of feasible partitions for a cyclic graph is always greater than the number of feasible partitions of one of its spanning trees -- a result proved in Appendix A. Consequently we can set a lower bound on the number of feasible partitions for a connected graph by finding the number of feasible partitions for the trees of Fig. 2.4. Fig. 2.4(a) represents the minimum-level k node tree and that of Fig. 2.4(b) the maximum-level k node tree.

In Appendix A we show that the number of feasible partitions of the minimum-level k node tree varies as $[f(W)]^k$ where $1 < f(W) < 2$ and f(W) is an increasing function of the weight constraint W. The minimum number of partitions of a tree of the form shown in Fig. 2.4(b) is $F_k \sim 1.6^k$ and occurs for a weight constraint of two. Here F_k is the kth Fibonacci number. An increase in the weight constraint results in an increase in the number of feasible partitions.

* A spanning tree of a graph G is a subgraph of G which is a tree that contains all nodes of G.



(a) Minimum-level k node tree



(b) Maximum-level k node tree

Figure 2.4 -- Minimum- and maximum-level k node trees

In conclusion, a k node connected graph has a number of feasible partitions that grows exponentially in k.

D. THE USE OF GRAPH PROPERTIES IN PARTITIONING

The computation and storage requirements of the dynamic programming procedure grow exponentially in k, limiting the utility of this procedure if it simply generates all feasible partitions. In this section we introduce several concepts that take advantage of properties of graphs. These properties significantly reduce the computation time and storage requirements for certain classes of graphs.

The first concept discussed is that of the isolated set. Using this concept we show that the growth in the number of partitions generated on step k of the partitioning process is dependent only on the degree of the nodes and not on the number of nodes k. The second concept takes into account the existence of cutpoints and blocks in a graph. In Chapter III we show that these two concepts form the basis of an efficient tree-partitioning algorithm.

A node i is defined to be an element of the isolated set for node k, denoted ISOL(k), if it satisfies the following properties:

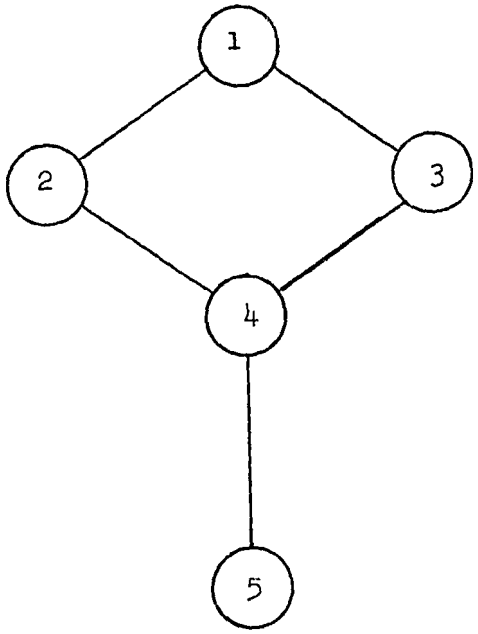
(1) The label i is less than k.

(2) Node i is not adjacent to any node with label $\geq$ k. Fig. 2.5 illustrates this definition.

Several properties of the isolated set result from this definition.

(1) The size of ISOL(k) is independent of the weight constraint.

(2) The connected set and the isolated set for any node k are mutually exclusive. This property follows from the definition of each set.



ISOL(1) = ϕ

ISOL(2) = ϕ

ISOL(3) = ϕ

ISOL(4) = {1}

ISOL(5) = {1,2,3}

Figure 2.5 -- Illustration of isolated set

(3) Let $CONN(k)_{max}$ denote the set of nodes with labels less than k that are not elements of ISOL(k). Then

$$|CONN(k)_{max}| = (k-1) - |ISOL(k)|$$

and is independent of the weight constraint.

(4) Given that CONN(k) represents the connected set of node k and ISOL(k) the isolated set,

$$ISOL(k) \subset \{1,2,\ldots,k-1\},$$

$$CONN(k) \subseteq \{1,2,\ldots,k-1\}.$$

Here $\{1,2,\ldots,k-1\}$ represents the set of nodes with labels less than k.

(5) $|CONN(k)_{max}| \geq |CONN(k)|$ for every weight constraint W. Note that $|CONN(k)|$ is a function of W and $|CONN(k)_{max}|$ is not.

The growth in the size of ISOL(k) is a nondecreasing function of k, as we show in the next theorem:

Theorem

$$|ISOL(k)| \leq |ISOL(k+1)|$$

Proof

Assume that $ISOL(k) \not\subseteq ISOL(k+1)$. Then there exists at least one node i that is in ISOL(k) but not in ISOL(k+1). By definition, i is adjacent to no node with label greater than k, consequently it is adjacent to no node with label greater than k+1, contrary to the assumption. Therefore, $ISOL(k) \subseteq ISOL(k+1)$. The value of k is finite, thus $|ISOL(k)| \leq |ISOL(k+1)|$. ■

We now show that the concept of the isolated set can be used to modify the partitioning process so that only a subset of the feasible partitions of a step of the process must be generated on that step.

Let the set of partitions of step k-1 be denoted by P_{k-1} and let the isolated set of node k be denoted by ISOL(k). The nodes in {1,2,...,k-1} not in ISOL(k) are denoted by $CONN(k)_{max}$. The set P_{k-1} can then be separated into disjoint subsets where the partitions in a given subset have the property below:

Let p and q be two partitions in the same subset of P_{k-1}. If $\{c_{ip}\}$ (for $i=1,2,\ldots,n_p$) denotes the n_p clusters of partition p and $\{c_{jq}\}$ (for $j=1,2,\ldots,n_q$) the n_q clusters of partition q, then for each cluster c_{ip} that contains nodes in $CONN(k)_{max}$, there is a cluster c_{jq} with equal weight that contains the same nodes of $CONN(k)_{max}$. An example of two partitions with this property is

$$p=(1,2)(4)(3,5) \text{ and } q=(1)(3)(4)(2,5)$$

where $CONN(6)_{max}=\{4,5\}$ and all nodes are of unit weight.

If a partition in a subset formed by this property has a cluster containing one or more nodes $i_1,i_2,\ldots,i_m$, each of which is in $CONN(k)_{max}$, then every other partition in the subset has a cluster of equal weight containing nodes $i_1,i_2,\ldots,i_m$.

Any two partitions in the same subset are defined as <u>similar partitions</u>. We define the <u>dominant partition</u> of a set of similar partitions as that partition of maximal value. If two or more partitions are similar, and have equal maximal values, then one is arbitrarily chosen as the dominant partition. The dominant partition is then said to "dominate" those partitions similar to it.

The reason for separating P_{k-1} into sets of similar partitions is that all but the dominant partition can be deleted from each subset of P_{k-1}. We show in Section E that this result reduces the upper bound on the number of feasible partitions generated on step k from

$$k! \doteq \sqrt{2\Pi k}\left(\frac{k}{e}\right)^{k}$$

to

$$\sqrt{2\Pi x_k^{3/2}}\left(\frac{x_k W}{e}\right)^{x_k},$$

where $x_k = |CONN(k)_{max}|$. For small values of W and x_k this result represents a significant reduction in the number of partitions that must be generated on the kth step of the partitioning process. We now prove that all but the dominant partitions of step k-1 can be deleted from P_{k-1}.

Isolated Set Theorem

The only partitions of step k-1 necessary in generating the partitions of step k are the dominant partitions.

Proof

Let G be an n node graph. A partition p generated on some step k in the process of partitioning G can be represented by a sequence of pairs

$$[1,(\ )],[2,c_2],[3,c_3],\ldots,[k,c_k],$$

where the first entry of a pair represents the node with label i and the second entry the cluster to which node i is added on step i. The advantage of this notation over the nodal representation is that it describes precisely how p is generated. An example of this notation is [1,( )], [2,( )],[3,(2)],[4,(1)],[5,(2,3)], where "( )" denotes the empty cluster. This representation is equivalent to the nodal representation p=(1,4)(2,3,5).

Let P_i be the set of partitions generated on step i of the partitioning process. We then define a derivation of a partition p from a partition q, where p is in P_k and q is in P_j $(j < k)$, as the sequence

$$[j+1,c_{j+1}],[j+2,c_{j+2}],\ldots,[k,c_k].$$

This notation is a variation of the above representation of p that ignores the steps leading up to the generation of partition q.

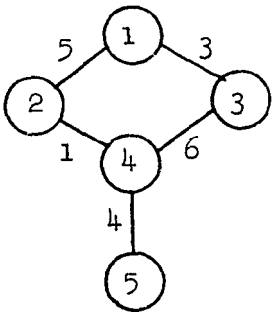
Let two partitions f and g generated on step k-1 be similar and let f dominate g. Assume that there exists a partition of G, g_n, derived from g that has a greater value than any partition of G derived from f. We now show that this assumption is false.

Let a derivation of g_n from g be $[k,c_k],[k+1,c_{k+1}],\ldots,[n,c_n]$. Since f and g are similar, there is a partition f_n derived from f with the derivation $[k,\bar{c}_k],[k+1,\bar{c}_{k+1}],\ldots,[n,\bar{c}_n]$ such that for i=k,k+1,...,n, c_i and $\bar{c}_i$ have the same weight and the nodes in c_i differ from those in $\bar{c}_i$ only if they are in ISOL(k). Note that the nodes in the isolated set of node k share no edge with a node whose label is greater than k-1. As a consequence, the values of partitions generated on steps k,k+1,...,n are independent of the nodes in ISOL(k) that appear in a cluster together with nodes in $\text{CONN}(k)_{\max}$.

Since clusters c_i and $\bar{c}_i$ (i=k,k+1,...,n) have nodes that differ only if they are in ISOL(k), the sum of the values of the edges in c_i and $\bar{c}_i$ can differ by the sum of the values of those edges between nodes in ISOL(k) contained in each cluster. Since f dominates g, the sum of the values of edges in $\bar{c}_i$ is equal to or greater than the sum of the edges in c_i and f_n dominates g_n. Consequently, the value of f_n is greater than or equal to the value of g_n, contrary to the assumption made above. It is therefore not contrary to an optimal policy to delete all partitions of P_{k-1} dominated by another partition. ■

An illustration of the results of this theorem is given in Fig. 2.6.

In Section 4 we generalize the reduction in growth of computation time and storage possible with the use of the Isolated Set Theorem.



$ISOL(5) = \{1,2,3\}$

$CONN(5)_{max} = \{4\}$

From Fig. 2.3, the sets of similar partitions in P_4 are:

S_1		S_2	
(1)(2)(3)(4)	VALUE=0	(1)(3)(2,4)	VALUE=1
(1,2)(3)(4)	VALUE=5	(1)(2)(3,4)	VALUE=6
(1,3)(2)(4)	VALUE=3	(1,3)(2,4)	VALUE=4
(1,2,3)(4)	VALUE=8	(1,2)(3,4)	VALUE=11

S_3	
(3)(1,2,4)	VALUE=6
(2)(1,3,4)	VALUE=9
(1)(2,3,4)	VALUE=7

The dominant partitions of P_4 are:

set	dominant partition	
S_1	(1,2,3)(4)	VALUE=8
S_2	(1,2)(3,4)	VALUE=11
S_3	(2)(1,3,4)	VALUE=9

Figure 2.6 -- Application of Isolated Set Theorem

The size of the isolated set for the nodes of a graph is a function of the labeling assigned to the graph. An analysis of the relationship between the labeling and the size of the isolated set is given in Chapter IV.

A cutpoint of a connected graph G=(V,E) is defined as a node c such that V-{c} is the node set of a nontrivial disconnected graph G'. A nonseparable graph is connected, nontrivial and has no cutpoints. A block of a graph G is a maximal nonseparable subgraph of G. An illustration of these definitions is given in Fig. 2.7.

If a connected graph G has more than one block, the following theorem proves that it is valid to find the optimal partitions of each block in any order and then combine these partitions to generate an optimal partition of G.

Theorem (Block Independence Theorem)

If a graph G has q blocks, where $q > 1$, then the optimal partition of G, p(opt), can be created by first partitioning the blocks independently, then combining the resulting partitions.

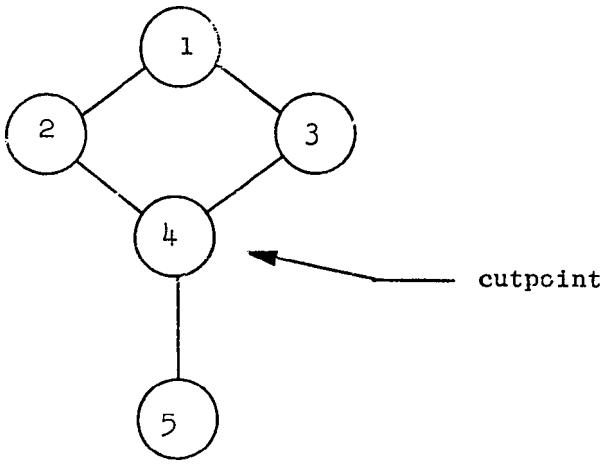
Proof

Consider the nodal representation of p(opt):

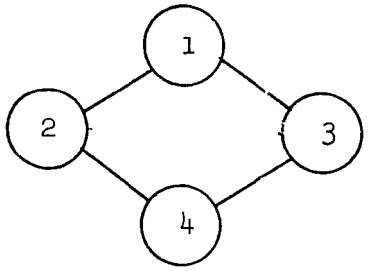
$$\underbrace{[(\ )\ldots(\ )]}_{NC_1}\underbrace{[(\ )\ldots(\ )]}_{NC_2}\ldots\underbrace{[(\ )\ldots(\ )]}_{NC_q}\underbrace{[(\ )\ldots(\ )]}_{C} .$$

Here, NC_i represents a (possibly empty) set of clusters whose nodes are not cutpoints and are all from the same block, B_i. The set C consists of clusters each of which contains at least one cutpoint.

The nodal representation of p(opt) assumes this form because of the special properties of a graph with one or more cutpoints. Since the only



"splitting" node 4 results in two blocks



block # 1

block # 2

Figure 2.7 -- Example of a graph with a cutpoint

node in a block B_i adjacent to nodes not in B_i is a cutpoint, a cluster that contains nodes from B_i, but no cutpoint, must only contain nodes from B_i as a result of the Connectivity Theorem. This property justifies the collection of clusters into sets NC_i for block B_i in the nodal representation above.

Each cluster $c \in C$ contains two types of nodes:

(1) a set of cutpoints $\{k_1,k_2,\ldots,k_x\}$;

(2) a set of nodes $\{i_1,i_2,\ldots,i_a,j_1,j_2,\ldots,j_b,..\}$, none of which are cutpoints.

The latter set can be partitioned into subsets by the equivalence relationship BLOCK, where u BLOCK v if u and v are nodes in the same block B_i. If the restriction on duplication of nodes implicit in the partitioning problem is removed, then the cluster c can be replaced by a set of clusters, $\{c_1,c_2,\ldots,c_z\}$, where these clusters have the following properties:

(1) each cluster c_i contains the union of the set of nodes of c from some block B_j created by the equivalence relation BLOCK and the set of cutpoints of c also in block B_j;

(2) $\sum_{i=1}^{z} \text{VALUE}[c_i] = \text{VALUE}[c]$, where $\text{VALUE}[c_i]$ equals the sum of the values of edges contained in cluster c_i.

Note that some cutpoint k may appear in several of the clusters making up the set $\{c_1,c_2,\ldots,c_z\}$.

When we perform the process above on each cluster in C, the nodal representation of p(opt) is transformed to

$$[\underbrace{(\ )\ldots(\ )}_{NC_1}]\ \ldots\ [\underbrace{(\ )\ldots(\ )}_{NC_q}]\ [\underbrace{(\ )\ldots(\ )}_{C_1}]\ \ldots\ [\underbrace{(\ )\ldots(\ )}_{C_q}],$$

where C_i=a set of clusters of nodes from block B_i including at least one cutpoint of B_i in each cluster. The value of the cover* p(opt)' given by this nodal representation equals that of p(opt), and is made up of sets of clusters (NC_i,C_i) representing a partition of block B_i. No edge exists from a cluster in the set (NC_i,C_i) to a cluster in the set (NC_j,C_j) for $i \neq j$ because of the duplication of cutpoints.

In conclusion we can reverse the process of decomposing p(opt) into the cover p(opt)' and generate p(opt) by first finding the partitions of each block, and then combining these partitions. ■

An implementation of the results of this theorem is given in Appendix C.

E. GROWTH RATE FOR GENERAL GRAPH PARTITIONING ALGORITHM

The following theorem develops an upper bound on the number of feasible partitions generated on the kth step of the partitioning process when modified to include the concept of the isolated set.

Theorem

Let $CONN(k)_{max}$ = the set of nodes with labels less than k not in ISOL(k), i.e.

$$CONN(k)_{max} = \{1,2,\ldots,k-1\} - ISOL(k),$$

and let

$$x_k = |CONN(k)_{max}|.$$

For a weight constraint of W there are no more than

$$x_k(x_k!)(W^{x_k}) \approx \sqrt{2\Pi x_k^{3/2}} \left[\frac{x_k W}{e}\right]^{x_k}$$

* A cover differs from a partition in that the intersection of the node sets of two clusters need not be empty.

partitions generated on step k of the partitioning process.

Proof

The partitions of step k-1 can be separated into disjoint subsets by the property that all partitions in a given subset have the same distribution of the nodes in CONN(k) in their clusters. If, for example, the set of partitions of step 4 is P_4={(1)(2)(3,4), (1,2)(3,4), (1,3)(2,4), and (1,2,3)(4)} and CONN(5)={3,4}, then the subsets of P_4 satisfying the above property are {(1,2)(3,4), (1)(2)(3,4)} and {(1,3)(2,4), (1,2,3)(4)}. Note that no limitation is placed upon the nodes in ISOL(k) in a cluster. We now show that any subset of P_{k-1} so formed has no more than

$$W^{x_k}$$

partitions in it, where W is the weight constraint and x_k is the maximum size of CONN(k) for any weight constraint.

Let P'_{k-1} be a set of partitions of step k-1 each of which has the same distribution of nodes in CONN(k) in its clusters. If a partition in P'_{k-1} has a cluster containing nodes $i_1,i_2,\ldots,i_m$ that are in CONN(k), then every other partition in P'_{k-1} also has a cluster containing nodes $i_1,i_2,\ldots,i_m$. No restriction is placed, however, on the nodes in ISOL(k) in a cluster containing this subset of CONN(k). Consequently, the weight of a cluster of a partition in P'_{k-1} containing nodes $i_1,i_2,\ldots,i_m$ need not be the same for each partition in P'_{k-1}. There are a maximum of x_k nodes in CONN(k), consequently we can distribute the nodes of CONN(k) into no more than x_k distinct clusters. Any given cluster can assume a weight that varies from one to W. Assume then that every partition in P'_{k-1} has x_k clusters that contain a node in CONN(k) and that every such cluster can have a weight that varies from one to W. The number of partitions in P'_{k-1} is then no greater than

$$W^{x_k}$$

since this number represents the number of different combinations of x_k clusters, where each cluster can assume a weight from one to W. This result follows from the Isolated Set Theorem, as we now show.

Assume that two partitions in P'_{k-1}, p and q, have clusters such that for every cluster of p containing a set of nodes in CONN(k), the cluster of q containing the same set of nodes in CONN(k) has equal weight. Also, assume that the value of p is greater than or equal to that of q. The Isolated Set Theorem then proves that q can be deleted from P'_{k-1}.

We now prove that an upper bound on the number of partitions of step k generated from the set P'_{k-1} is given by

$$x_k W^{x_k}$$

where for simplicity we assume that $W \leq x_k$.

Assume that each partition in the set P'_{k-1} has r clusters that contain at least one node in the set CONN(k). Also, let each node have unit weight. Node k can then be added to each of the r clusters of a partition in P'_{k-1} if the weight of the cluster to which k is added is less than W. Let P(i) denote the set of partitions in P'_{k-1} whose ith cluster, of those clusters that contain a node in CONN(k), has weight less than W. The number of feasible partitions of step k generated by adding node k to a cluster of a partition of P'_{k-1} is then given by

$$\sum_{i=1}^{r} |P(i)| \ .$$

The upper bound on $|P(i)|$ is given by

$$|P(i)| \leq W^{r-1}(W-1),$$

and the maximum value of r is x_k, therefore no more than

$$x_k W^{x_k-1}(W-1)$$

partitions of step k can result from adding node k to the clusters of the partitions in P'_{k-1}. There are W^{x_k} 1-adjacencies of step k derived from the partitions in P'_{k-1}, hence

$$x_k(W-1)W^{x_k-1} + W^{x_k}$$

partitions are generated from the subset P'_{k-1}. If we assume that $W \leq x_k$ then

$$x_k(W-1)W^{x_k-1} + W^{x_k} \leq x_k W^{x_k}.$$

From Section C there are less than $x_k!$ possible ways to distribute the nodes in CONN(k) in clusters, hence the set P_{k-1} can be separated into no more than $x_k!$ subsets. Therefore the upper bound on the number of partitions generated on step k of the partitioning algorithm is

$$x_k(x_k!)W^{x_k},$$

where x_k is independent of the weight constraint.

If the dynamic programming procedure were not modified to take into account the existence of isolated nodes, the growth in the size of P_k is exponential, ranging from y^k, where $1 < y < 2$, for the simple trees of Fig. 2.4 to k! for total enumeration. The growth in P_k for an algorithm consisting of the dynamic programming procedure and a procedure for deleting suboptimal partitions based upon the concept of the isolated set has an upper bound of

$$x_k f(x_k) W^{x_k} \quad \text{where } 1 \leq f(x_k) < x_k! .$$

(The lower bound of $f(x_k!)$ occurs for the graph of Fig. 2.8). If x_k and W are small, a significant reduction in the size of P_k results from the use of the concept of the isolated set.

To illustrate the effectiveness of the isolated set in reducing the partitions generated on each step of the partitioning process, we now examine several graph types that readily lend themselves to analysis.

A dramatic example of the reduction in computation time and storage is the following. In Section C we show that the minimum number of partitions generated on the kth step for the simple k node tree of Fig. 2.4(b) is greater than 1.6^k. Using the analysis above this bound is reduced to the following:

$$|P_k| \leq x_k(x_k!)W^{x_k} \quad \text{where } x_k=1 \text{ for all } k > 1$$

thus

$$|P_k| \leq W.$$

Another graph whose value of x_k is independent of k is that of Fig. 2.8. For a width parameter h, ISOL(k)={i|i has label < k-h}. Thus

$$x_k=h \quad \text{for all k and}$$
$$|P_k| \leq h(h!)W^h$$

A more careful analysis results in the upper bound

$$|P_k| \leq W^h .$$

A graph with a constant fan-out f is the fully developed tree, an example of which is shown in Fig. 2.9. Lawler, Levitt, and Turner [Goldberg, et al., 1967] have shown that the growth for W=2 in the number of feasible partitions for a fully-developed k node tree with fan-out of f is bounded by

$$|P_k(W=2)| < 2^{f+1}(f+1)^k.$$

In the following theorem we show that this bound can be reduced by employing the concept of the isolated set.

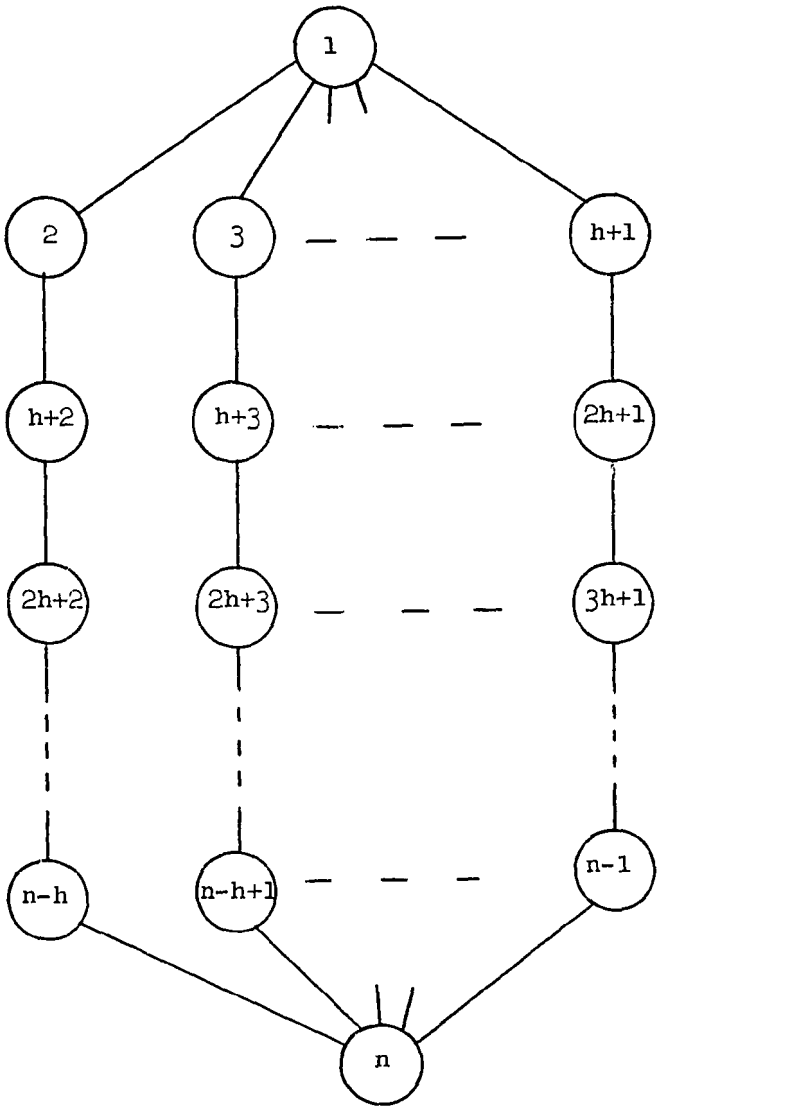


Figure 2.8 -- Graph with constant-size connected set

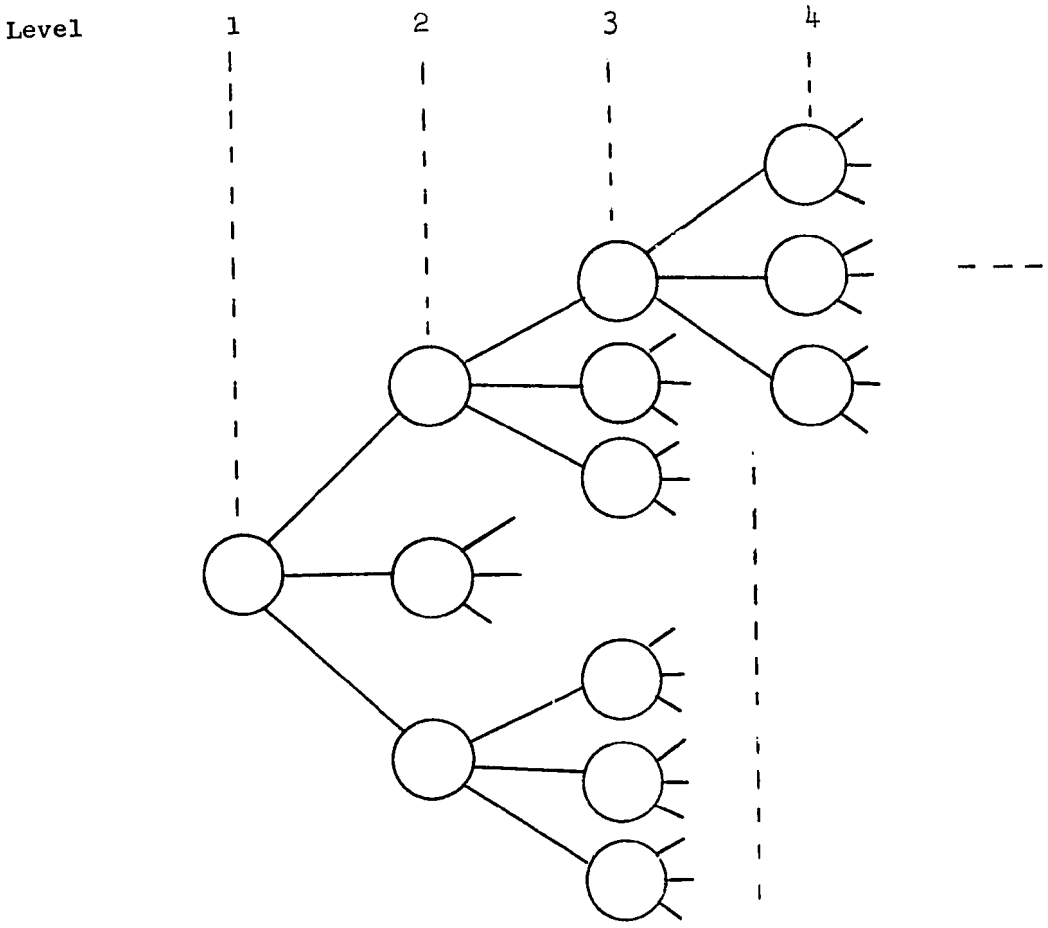


Figure 2.9 -- Fully-developed tree

Theorem

A fully-developed k node tree with fan-out of f has an upper bound of

$$(f+1)^{k/f}$$

feasible partitions for a weight constraint of two.

Proof

Assume that the tree is labeled such that node 1 is the root, nodes 2,3,...,f+1 are at level 1, nodes f+2,f+3,...,2f+1 are at level 2, etc. Given any node j at level x of the tree, all nodes at levels 1,2,...,x-2 are in ISOL(k). A partition of step k then assumes the form

$$[\underbrace{(\ )(\ )\ldots(\ )}_{A}][\underbrace{(\ )(\ )\ldots(\ )}_{B}]$$

where "( )" denotes a cluster of the partition. Set A comprises clusters all of whose nodes are in ISOL(k). Set B consists of clusters containing nodes at levels x-1 and x. The number of feasible partitions of the tree is then the number of possible distributions of nodes in clusters in the set B.

For W=2 a feasible cluster in set B contains either a single node at level x-1 or x, or a node at level x and its predecessor at level x-1. Separate the nodes at levels x and x-1 into subsets $S_1,S_2,\ldots,S_y$, where the nodes in the same subset consist of:

(1) the nodes at level x with the same predecessor node i, and

(2) the common predecessor node i.

The number of subsets y equals the number of nodes at level x-1, therefore $y=f^{x-2}$.

Each subset S_i contains f+1 nodes. These nodes can form no more than f+1 feasible partitions for W=2 since a cluster of a feasible

partition that contains more than one node must contain the node in S_i at level x-1. The distribution of the f+1 nodes in each of the y subsets $S_1, S_2, \ldots, S_y$ is independent of the distribution in the other subsets, consequently there are

$$(f+1)^y$$

possible ways to cluster the nodes at levels x and x-1. The value of k is related to the level x by

$$k = \frac{(f^x-1)}{(f-1)}$$

therefore

$$f^{x-2} = \frac{k}{f} - \frac{k}{f^2} + \frac{1}{f^2} < k/f \quad (k>1).$$

The upper bound on the number of feasible partitions for the tree is therefore

$$|P_k| < (f+1)^{k/f} \quad \text{for } W=2. \blacksquare$$

In Chapter III we show that any tree can be partitioned with a total number of operations directly proportional to the number of nodes in the tree.

In conclusion, the introduction of the concept of the isolated set bounds the number of partitions generated on the kth step of the dynamic programming procedure to a maximum of

$$|P_k| \leq x_k f(x_k) W^{x_k} \quad \text{where}$$

$$x_k = |CONN(k)_{max}|$$

$$= (k-1) - |ISOL(k)| \text{ and}$$

$$1 \leq f(x_k) < x_k!$$

An implementation of the result of the Block Independence Theorem is given in Appendix C. We show there that the maximum number of partitions generated on any step is directly proportional to the number of partitions generated if each block were partitioned independently. In many cases this reduces the growth in the cardinality of P_k from an exponential in k to an exponential in k', where $k' \ll k$.

F. THE GENERAL GRAPH PARTITIONING ALGORITHM

The concepts of block independence and the isolated set introduced in Section D can be combined with the dynamic programming procedure to form an algorithm for partitioning a general graph with a substantial improvement in the growth in computational and storage requirements. Section E has shown this improvement.

The general partitioning algorithm is summarized in Figs. 2.10 and 2.11. These flow charts consist of three procedures:

(1) A procedure that determines the blocks in a graph and then generates the partitions of these blocks, combining them with partitions of other blocks of the graph.

(2) A basic partitioning algorithm consisting of two subprocedures:

   (a) A dynamic programming procedure that generates feasible partitions.

   (b) A procedure, based upon the concept of the isolated set, that deletes all but the dominant partitions on each step of the dynamic programming procedure.

The flow chart shown in Fig. 2.10 contains a procedure (A) to find the blocks of a given graph. This algorithm is outlined in Hopcroft and Tarjan [1971]. If no cutpoints exist in the graph to be partitioned,

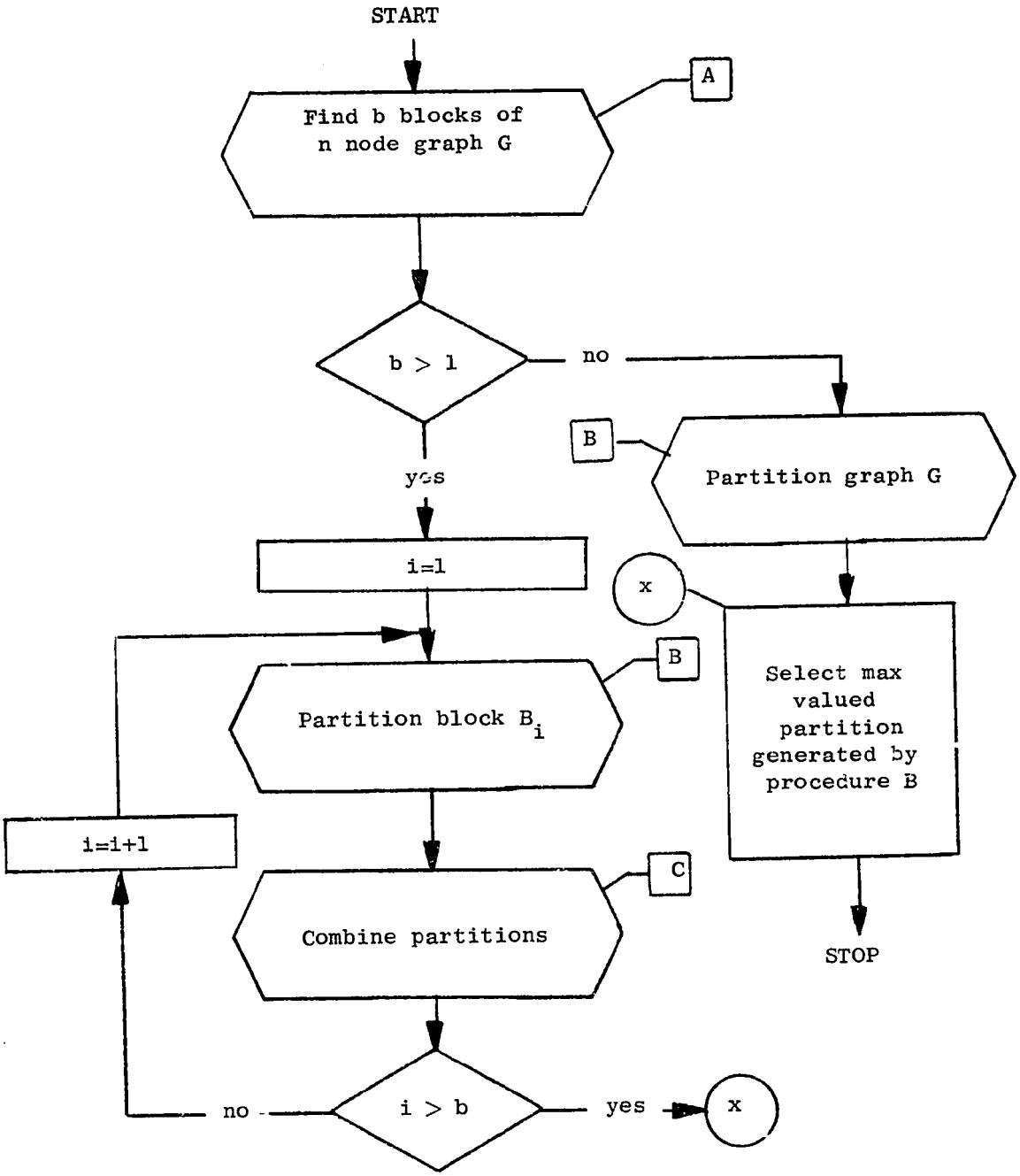


Figure 2.10 -- Flowchart of general graph partitioning process

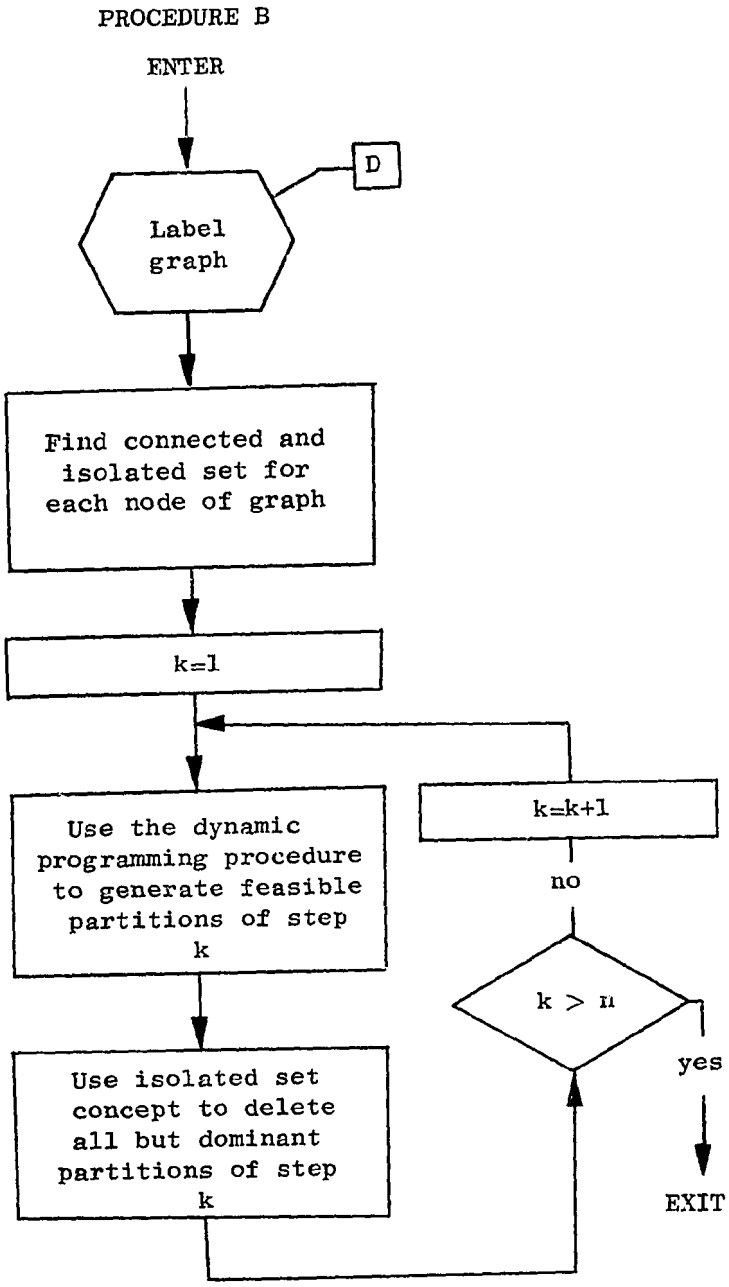


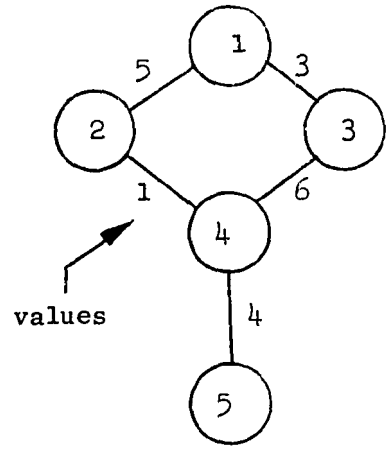
Figure 2.11 -- Flowchart of basic partitioning algorithm

then the basic partitioning algorithm (procedure B) is performed and the resulting maximal-valued partition is the optimal partition of the graph.

If more than one block exists in the graph each block is partitioned and the results combined with the partitions of other blocks to form the optimal partition of the graph. An implementation of procedure C that performs this task is given in Appendix C.

An implementation of the basic partitioning algorithm is given in Appendix B. The procedure (D) for labeling the graph is the subject of Chapter IV.

An example of the use of the basic partitioning algorithm is given in Fig. 2.12. It is instructive to compare the number of partitions generated here with the number generated using the dynamic programming procedure alone (Fig. 2.3). We see that significantly fewer partitions are generated on each step by the general partitioning algorithm. We have not made use of the Block Independence Theorem, although the graph has two blocks. An example of the use of this theorem is given in Appendix C.



all nodes unit weight
W = 3

k	CONN(k)	ISOL(k)
1	ϕ	ϕ
2	{1}	ϕ
3	{1,2}	ϕ
4	{2,3}	{1}
5	{4}	{1,2,3}

k-adjacencies

STEP	1	2	3
1	(1) = 0		
2	(1)(2) = 0	(1,2) = 5	
3	(1)(2)(3) = 0 (1,2)(3) = 5	(1,3)(2) = 3 (1)(2,3) = 0	(1,2,3) = 8
4	(1,2,3)(4) = 8	(1)(2,4)(3) = 1 (1)(2)(3,4) = 6 (1,2)(3,4) = 11	(1,2,4)(3) = 6 (1,3,4)(2) = 9 (1)(2,3,4) = 7
5	(1,2)(3,4)(5) = 11	(1,2,3)(4,5) = 12	(1,2)(3,4,5) = 15

optimal partition is (1,2)(3,4,5) with VALUE = 15

Figure 2.12 -- Example of graph partitioning algorithm

CHAPTER III

AN EFFICIENT TREE-PARTITIONING ALGORITHM

A very efficient variation of the general algorithm described in Chapter II results if the graph to be partitioned is a tree. The concepts of the isolated set and block independence, combined with the property that a tree has no cycles, result in a partitioning algorithm whose growth in computation time is directly proportional to the number of nodes in the tree.

Before describing this algorithm, we note that the ability to partition a tree with integer-weighted nodes and multi-valued edges has not been considered in the literature. Kernighan [1969] describes an algorithm that partitions a tree with a growth in computation of $n(\log_2 n)$ for an n node tree. The edges of this tree must, however, assume a restricted set of values.

A. INTRODUCTION

A rooted tree is a directed graph T with node set V containing one or more nodes such that:

(1) there is a specially designated node of V called the root of T, and

(2) the remaining nodes in V can be separated into $m \geq 0$ disjoint subsets $V_1, V_2, \ldots, V_m$ such that each V_i is the node set of a rooted tree T_i $(i=1,2,\ldots,m)$. The trees $T_1, T_2, \ldots, T_m$ are called the subtrees of the root.

If the relative order of the subtrees $T_1, T_2, \ldots, T_m$ is important, the tree is an ordered tree. The degree of a node of the rooted tree equals the number of subtrees of that node. A leaf has degree zero and a branch node has degree greater than zero. The roots of the subtrees of a branch node k are the sons of node k.

The tree partitioning algorithm of Section B can only partition an ordered tree. We show in Sections B and C that the particular ordered tree employed has no effect upon the growth in computation time and storage space requirements of the algorithm. The graph to be partitioned G is therefore transformed into an ordered tree G' by the following procedure:

(1) Assign a unique label to each node in G.

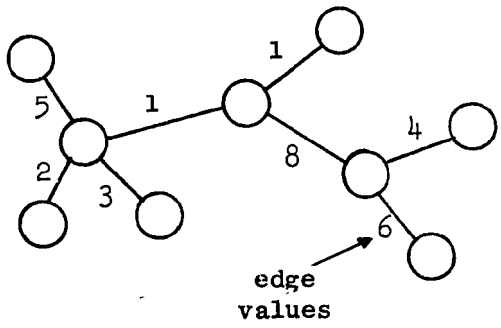
(2) Form a rooted tree by selecting any node of G as root.

(3) Order the subtrees of a branch node k by increasing label of their roots.

Whereas the growth in computation time and storage of the general graph partitioning algorithm is a function of the labeling, such is not the case for the algorithm described here; the labels assigned to nodes in step 1 above are merely identifiers. An example of the transformation of a tree G to an ordered tree G' is shown in Fig. 3.1. Before describing the tree-partitioning algorithm, we introduce the following notation. In this notation small letters represent partitions and capital letters represent subgraphs. In particular, the letter q represents a partition of a subtree of the ordered graph G', p represents a partition of a subgraph of G', and S represents a subgraph of G'. We also use the shorter term "subtree k" to mean the subtree of G' whose root is the node with label k.

$\underline{q(k,w) \text{ and } \bar{q}(k)}$:

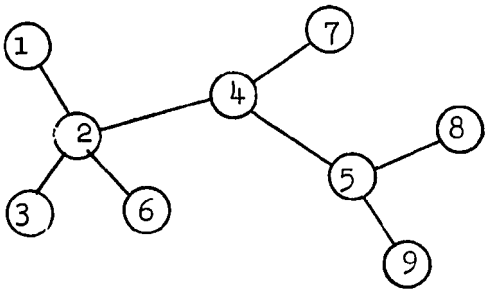
The maximal-valued partition of subtree k whose cluster containing node k is of weight w is denoted by q(k,w). The maximal-valued partition of subtree k for all weights is denoted by $\bar{q}(k)$. Fig. 3.2 illustrates these definitions.



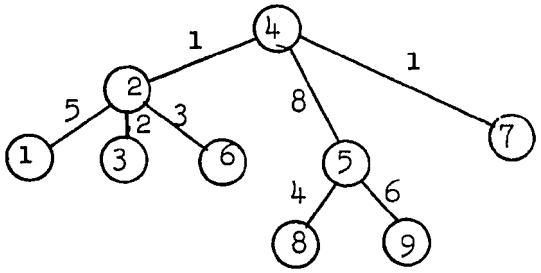
all nodes unit weight

W = 3

(a) Graph to be partitioned, G

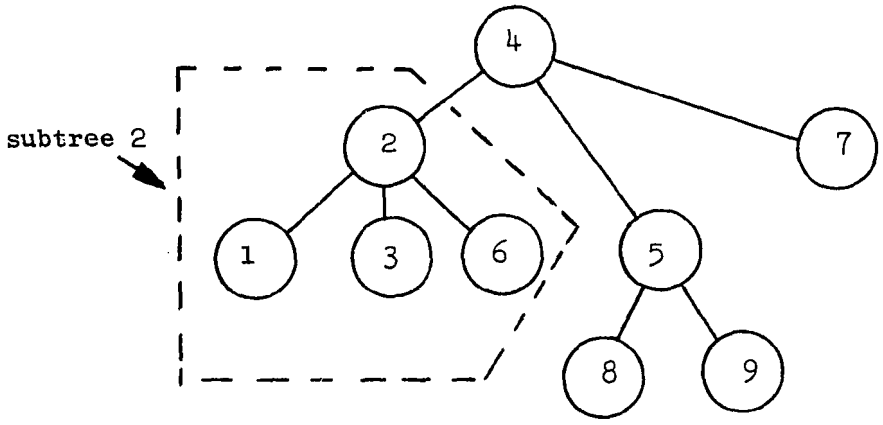


(b) A unique label is assigned to each node of G

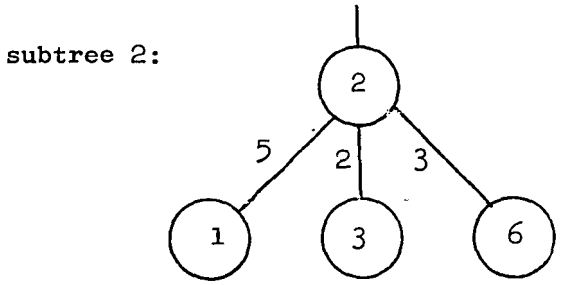


(c) Ordered tree G'

Figure 3.1 -- Transformation of tree G into ordered tree G'



(a) An illustration of the notation "subtree k"



let all nodes have unit weight and W = 3

Then q(2,1)=(1)(2)(3)(6) VALUE=0

q(2,2)=(1,2)(3)(6) VALUE=5 $\bar{q}(2)=q(2,3)$

q(2,3)=(1,2,6)(3) VALUE=8

(b) An illustration of the notation q(k,w)

Figure 3.2 -- Illustration of notation

$S_i(k)$:

Let $S_i(k)$ denote the subgraph of G' with the following properties:

(1) The node set of $S_i(k)$ consists of the branch node k and the nodes in the first i subtrees of node k.

(2) The edge set of $S_i(k)$ consists of the edges from node k to the first i sons of k and the edges in the first i subtrees of node k.

$p_i(k,w)$:

The tree-partitioning algorithm iteratively generates partitions of $S_{i+1}(k)$ by combining the partitions of $S_i(k)$ with the partitions of the i+1 st subtree of node k. The maximal-valued partition of $S_i(k)$ whose cluster containing node k is of weight w is denoted by $p_i(k,w)$. Fig. 3.3 illustrates this notation.

$[p_i(k,w_1),q(j,w_2)]$ and $[p_i(k,w)][\bar{q}(j)]$:

If subtree j is the i+1 st subtree of branch node k, denote by

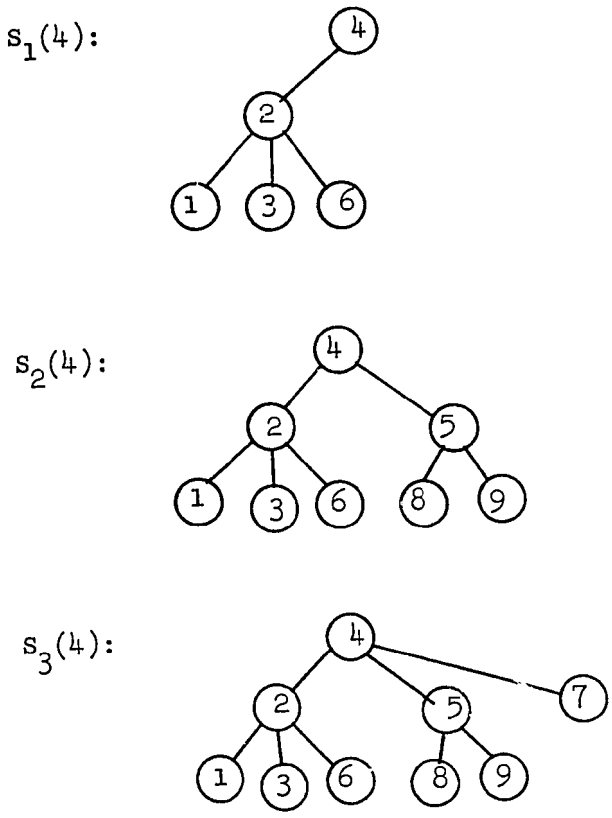
$$[p_i(k,w_1),q(j,w_2)]$$

the partition of $S_{i+1}(k)$ whose cluster containing node k is of weight w_1+w_2. The set of clusters of the partition so represented contains one cluster created by merging the cluster of $p_i(k,w_1)$ containing node k and the cluster of $q(j,w_2)$ containing node j. The other clusters of $[p_k(k,w_1),q(j,w_2)]$ are made up of the remaining (unmodified) clusters of $p_i(k,w_1)$ and $q(j,w_2)$.

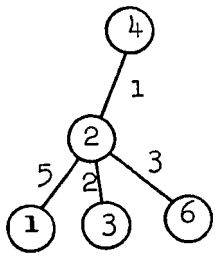
Denote by

$$[p_i(k,w)][\bar{q}(j)]$$

the partition of $S_{i+1}(k)$ whose cluster containing node k has weight w



(a) An illustration of the notation $S_i(k)$



if each node has unit weight and W=3 then

$p_1(4,1)=(1,2,6)(3)(4)$ VALUE=8

$p_1(4,2)=(2,4)(1)(3)(6)$ VALUE=1

$p_1(4,3)=(1,2,4)(3)(6)$ VALUE=6

(b) An illustration of the notation $p_i(k,w)$

Figure 3.3 -- Illustrations of notation

created by concatenating the unmodified clusters of $p_i(k,w)$ and $\bar{q}(j)$. Fig. 3.4 illustrates the use of this notation.

B. ALGORITHM

STEP 1

Form an ordered tree G' using the method given in Section A.

STEP 2

Initialize every leaf k of G' such that

$$q(k,w)=(k) \text{ and } VALUE[q(k,w)]=0.$$

Also

$$\bar{q}(k)=(k) \text{ and } VALUE[\bar{q}(k)]=0.$$

STEP 3

Find a branch node k all of whose sons are leaf nodes. If no such node exists, go to step 5.

STEP 4

If node k has m sons with labels $j_1,j_2,\ldots,j_m$, find the partitions of subtree k as follows (for w=WEIGHT[k], WEIGHT[k]+1,...,W):

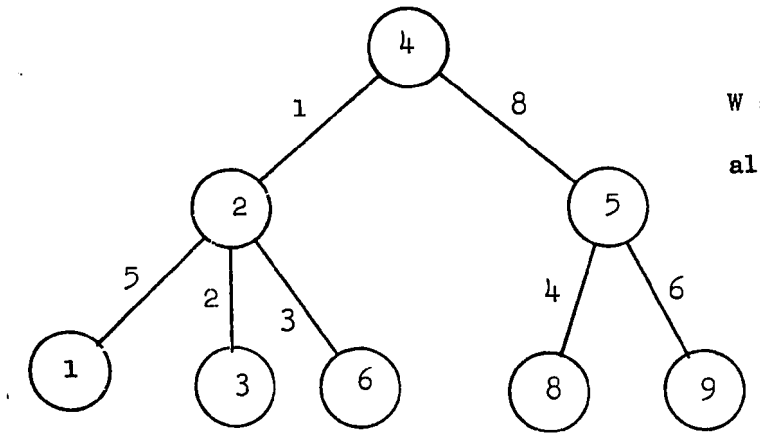
(a) Let i=1 and $p_0(k,w)=(k)$ if w=WEIGHT[k] where $VALUE[p_0(k,w)]=0$.

(b)
$$p_1(k,w)=\begin{cases}[p_0(k,w)][\bar{q}(j_1)] \text{ if } w=WEIGHT[k], \text{ or}\\ [p_0(k,w_1),q(j_1,w_2)]\\ \quad \text{if } w > WEIGHT[k]\end{cases}$$

where w_1=WEIGHT[k] and w_2=w-WEIGHT[k] such that $w_2 \geq WEIGHT[j_1]$.

$$VALUE[p_1(k,w)]=\begin{cases}VALUE[\bar{q}(j_1)] \text{ if } w=WEIGHT[k], \text{ or}\\ VALUE[q(j_1,w_2)]+VALUE[edge(k,j_1)]\\ \quad \text{if } w\neq WEIGHT[k].\end{cases}$$

Delete partitions of subtree j_1 from storage.



W = 3

all nodes unit weight

If $q(5,1)=(5)(8)(9)$ and $p_1(4,1)=(4)(1,2,6)(3)$

$q(5,2)=(5,9)(8)$ $p_1(4,2)=(2,4)(1)(3)(6)$

$q(5,3)=(5,8,9)$ $p_1(4,3)=(1,2,4)(3)(6)$

$\bar{q}(5)=q(5,3)$

then

$$[p_1(4,2),q(5,1)]=(2,4,5)(8)(9)(1)(3)(6)$$

and

$$[p_1(4,3)][\bar{q}(5)]=(1,2,4)(3)(6)(5,8,9)$$

Figure 3.4 -- Illustration of notation

(c) i=i+1

$p_i(k,w)$ = maximal-valued partition of the collection:

$$\begin{cases} [p_{i-1}(k,w)][\bar{q}(j_i)] \text{ and} \\ [p_{i-1}(k,w_1),q(j_i,w_2)] \text{ with} \end{cases}$$

w_1=WEIGHT[k],WEIGHT[k]+1,...,w-WEIGHT[j_i]

and w_2=w-w_1 such that $w_2 \geq$ WEIGHT[j_i].

Here

$$\text{VALUE}\{[p_{i-1}(k,w)][\bar{q}(j_i)]\} = \text{VALUE}[p_{i-1}(k,w)] + \text{VALUE}[\bar{q}(j_i)],$$

and

$$\text{VALUE}\{[p_{i-1}(k,w_1),q(j_i,w_2)]\} = \text{VALUE}[p_{i-1}(k,w_1)] + \text{VALUE}[q(j_i,w_2)] + \text{VALUE}[\text{edge}(k,j_i)].$$

(d) Delete the partitions of subtree j_i from storage.

(e) If i=m then:

(i) $q(k,w)=p_m(k,w)$

(ii) $\bar{q}(k)=p_m(k,w)$ such that VALUE[$p_m(k,w)$] is maximal for w=WEIGHT[k],WEIGHT[k]+1,...,W.

(iii) Prune nodes $j_1,j_2,\ldots,j_m$ from G'.

(iv) Store all q(k,w), $\bar{q}(k)$, and go to step 3.

If i≠m go to (c).

STEP 5

If r is the label of the root node, then $\bar{q}(r)$ is the optimal-valued partition of the given tree G.

The efficiency of the tree-partitioning algorithm is due to the ability to perform global optimization through local operations. The algorithm is based upon the following theorem.

Theorem

Let a set of feasible partitions of subgraph $S_i(k)$ be separated into disjoint subsets where all partitions in the same subset have a cluster of the same weight that contains node k. Then, all but a maximal-valued partition from each subset can be deleted. The resulting maximal-valued partition, whose cluster containing node k is of weight w, is denoted by $p_i(k,w)$, where w=WEIGHT[k],WEIGHT[k]+1,...,W.

Proof

All nodes of $S_i(k)$, with the exception of node k, are adjacent to no node in the subgraph of G' yet to be partitioned. As a consequence, the connectivity constraint dictates that the only cluster of a partition of $S_i(k)$ modified in future steps of the partitioning process is that containing node k. We can use an argument identical to that used in the proof of the Isolated Set Theorem to show that, of the set of partitions of $S_i(k)$ with a cluster of weight w containing node k, all but the maximal-valued partition can be deleted from further consideration in the partitioning process. ■

Corollary

Given a set of feasible partitions of subtree k, all but the maximal-valued partitions with a cluster containing node k of weight w for w=WEIGHT[k], WEIGHT[k]+1,...,W can be deleted from further consideration in the partitioning process. The maximal-valued partition of subtree k whose cluster containing node k is of weight w is denoted by q(k,w).

Proof

Since $S_m(k)$ represents subtree k if node k has m sons, and $q(k,w)=p_m(k,w)$, this result is a special case of the above theorem. ■

We now prove the optimality of the tree-partitioning algorithm.

Theorem

The tree-partitioning algorithm generates the optimal-valued partition of the given tree, G.

Proof

Let G' be the ordered tree to which tree G is transformed in step 1 of the algorithm. The Block Independence Theorem proves that the optimal-valued partition of a graph with more than one block can be created by generating the partitions of each block independently and then combining these partitions. We can extend this result to the practice of generating the partitions of disjoint subgraphs containing more than one block and then combining the resulting partitions to form the optimal partitions of the graph. Since every subtree of the ordered tree G' represents a collection of blocks of G, the generation of the optimal partition of G' can be performed by first generating the partition of each subtree whose root is a son of a branch node k, and then combining these partitions in any order to create the feasible partitions of subtree k. Note that this result justifies the assumption that the order in which the subtrees of each branch node k are combined is unimportant. In Section C we further show that the particular rooted tree used to form G' from G has no effect upon the partitioning algorithm.

We now show that the method used to generate the optimal-valued partitions of each subtree is correct.

The previous theorem and its corollary prove that all but a maximal-valued representative of the partitions of subgraph $S_{i-1}(k)$ whose cluster containing node k has weight w can be deleted. The proof of this theorem is based upon the fact that the only cluster of a partition of $S_{i-1}(k)$ modified when that partition is combined with partitions of other sub-

graphs is the cluster containing node k. Consequently, when forming the partitions of $S_i(k)$ we need only consider the possible combinations of the cluster of a partition of subtree j_i containing node j_i and the cluster of a partition of subgraph $S_{i-1}(k)$ containing node k. Here node j_i is the ith son of node k. The collection

$$[p_{i-1}(k,w)][\bar{q}(j_i)] \text{ and}$$

$$[p_{i-1}(k,w_1),q(j_i,w_2)]$$

$$\text{with } w_1=\text{WEIGHT}[k],\text{WEIGHT}[k]+1,\ldots,w-\text{WEIGHT}[j_i]$$

$$\text{and } w_2=w-w_1 \text{ such that } w_2 \geq \text{WEIGHT}[j_i]$$

then enumerates the ways in which two clusters, one containing node k and the other node j_i, can be combined to result in a cluster of weight w containing node k. ■

C. COMPUTATIONAL AND STORAGE GROWTH RATES

Consider a step in which the partition $p_i(k,w)$ is generated. There are a maximum of w ways to form $p_i(k,w)$ since the collection of partitions from which $p_i(k,w)$ is selected is enumerated by

$$[p_{i-1}(k,w)][\bar{q}(j_i)] \text{ and}$$

$$[p_{i-1}(k,w_1),q(j_i,w_2)] \text{ where } w_1 \text{ ranges from a minimum of 1}$$

$$\text{to a maximum of w-1 and } w_2=w-w_1 \text{ such that } w_{2_{min}}=1.$$

Since w can range from one to W, there are W(W+1)/2 partitions generated on each iteration of the step that combines the partitions of $S_{i-1}(k)$ and the partitions of the ith subtree of k, subtree j_i. For a root with p sons, there are p iterations of this step, hence

$$W(W+1)p/2$$

operations per root node. The sum of the number of sons for all roots in

any rooted tree is equal to (n-1) for an n node tree, therefore the growth in computational complexity for this algorithm is

$$\frac{W(W+1)(n-1)}{2} \approx W^2 n.$$

Note that this growth rate is dependent only on the number of nodes in the tree, not the particular rooted tree used to represent the tree.

At any point in the partitioning process, the maximum amount of storage required occurs if some node has p subtrees and each subtree has W partitions, hence a maximum of less than nWM words of storage are required, where M represents the number of words of storage required to store a partition.

In conclusion, the storage and computational requirements of the tree-partitioning algorithm are linear in the number of nodes in the tree.

D. EXAMPLE

We now illustrate the use of the algorithm by partitioning the graph of Fig. 3.1(a).

STEP 1

The transformation of the graph of Fig. 3.1(a) to an ordered tree is outlined in Fig. 3.1.

STEP 2

Initialize: $\overline{q}(k)=q(k,1)=(k)$

VALUE=0

where k=1,3,6,7,8,9

STEP 3 (iteration 1)

Select branch node 2 with sons 1,3, and 6.

STEP 4 (iteration 1)

(a) i=1 and $p_0(2,1)=(2)$ VALUE=0

(b) $p_1(2,1)=[p_0(2,1)][\bar{q}(1)]=(1)(2)$ VALUE=0

$p_1(2,2)=[p_0(2,1),q(1,1)]$

$=(1,2)$ VALUE=5

$p_1(2,3)$ does not exist

Delete partitions of subtree 1.

(c) i=2

Select $p_2(2,1)$ from:

$\boxed{[p_1(2,1)][\bar{q}(3)]=(1)(2)(3) \quad \text{VALUE=0}}$ *

Select $p_2(2,2)$ from:

$\boxed{[p_1(2,2)][\bar{q}(3)]=(1,2)(3) \quad \text{VALUE=5}}$

$[p_2(2,1),q(3,1)]=(2,3)(1)$ VALUE=2

Select $p_2(2,3)$ from:

$[p_1(2,3)][\bar{q}(3)]$ does not exist

$\boxed{[p_1(2,2),q(3,1)]=(1,2,3) \quad \text{VALUE=7}}$

$[p_1(2,1),q(3,2)]$=does not exist

(d) Delete partitions of subtree 3.

(e) $i\neq 3$, therefore go to (c)

(f) i=3

Select $p_3(2,1)$ from:

$\boxed{[p_2(2,1)][\bar{q}(6)]=(1)(3)(2)(6) \quad \text{VALUE=0}}$

* Boxed partition is maximal-valued partition in collection.

Select $p_3(2,2)$ from:

$[p_2(2,2)][\bar{q}(6)]=(1,2)(3)(6)$ VALUE=5

$[p_2(2,1),q(6,1)]=(1,6)(2)(3)$ VALUE=3

Select $p_3(2,3)$ from:

$[p_2(2,3)][\bar{q}(6)]=(1,2,3)(6)$ VALUE=7

$[p_2(2,2),\ q(6,1)]=(1,2,6)(3)$ VALUE=8

$[p_2(2,1),q(6,2)]$=does not exist

(d,e) Delete partitions of subtree 6.

Since i=3, let

$q(2,1)=p_3(2,1)=(1)(2)(3)(6)$ VALUE=0

$q(2,2)=p_3(2,2)=(1,2)(3)(6)$ VALUE=5

$\bar{q}(2)=q(2,3)=p_3(2,3)=(1,2,6)(3)$ VALUE=8

Prune nodes 1,3, and 6 from G'.

STEP 3 (iteration 2)

Select branch node 5 with sons 8 and 9.

STEP 4 (iteration 2)

A summary of the results of step 4 is:

$q(5,1)=(5)(8)(9)$ VALUE=0

$q(5,2)=(5,9)(8)$ VALUE=6

$q(5,3)=(5,8,9)$ VALUE=10

$\bar{q}(5)=q(5,3)$

STEP 3 (iteration 3)

Select root node 4 with sons 2,5, and 7.

STEP 4 (iteration 3)

(a) i=1 and $p_0(4,1)=(4)$ VALUE=0

(b) $p_1(4,1)=[p_0(4,1)][\bar{q}(2)]=(4)(1,2,6)(3)$ VALUE=8

$p_1(4,2)=[p_0(4,1),q(2,1)]=(1)(2,4)(3)(6)$ VALUE=1

$p_1(4,3)=[p_0(4,1),q(2,2)]=(1,2,4)(3)(6)$ VALUE=6

Delete partitions of subtree 2 from storage.

(c) i=2

Select $p_2(4,1)$ from:

$[p_1(4,1)][\bar{q}(5)]=(4)(1,2,6)(3)(5,8,9)$ VALUE=18

Select $p_2(4,2)$ from:

$[p_1(4,2)][\bar{q}(5)]=(1)(2,4)(3)(6)(5,8,9)$ VALUE=11

$[p_1(4,1),q(5,1)]=(4,5)(8)(9)(1,2,6)(3)$ VALUE=16

Select $p_2(4,3)$ from:

$[p_1(4,3)][\bar{q}(5)]=(1,2,4)(3)(6)(5,8,9)$ VALUE=16

$[p_1(4,2),q(5,1)]=(1)(2,4,5)(8)(9)(3)(6)$ VALUE=9

$[p_1(4,1),q(5,2)]=(4,5,9)(1,2,6)(3)(8)$ VALUE=22

(e) $i\neq 3$, therefore go to step (c)

(c,d,e) Summary of these steps:

$q(4,1)=(1,2,6)(3)(5,8,9)(4)(7)$ VALUE=18

$q(4,2)=(1,2,6)(3)(4,5)(8,9)(7)$ VALUE=16

$q(4,3)=(1,2,6)(3)(8)(4,5,9)(7)$ VALUE=22

STEP 3,5

Maximal-valued partition of tree is $\bar{q}(4)=q(4,3)$. This partition is shown in Fig. 3.5.

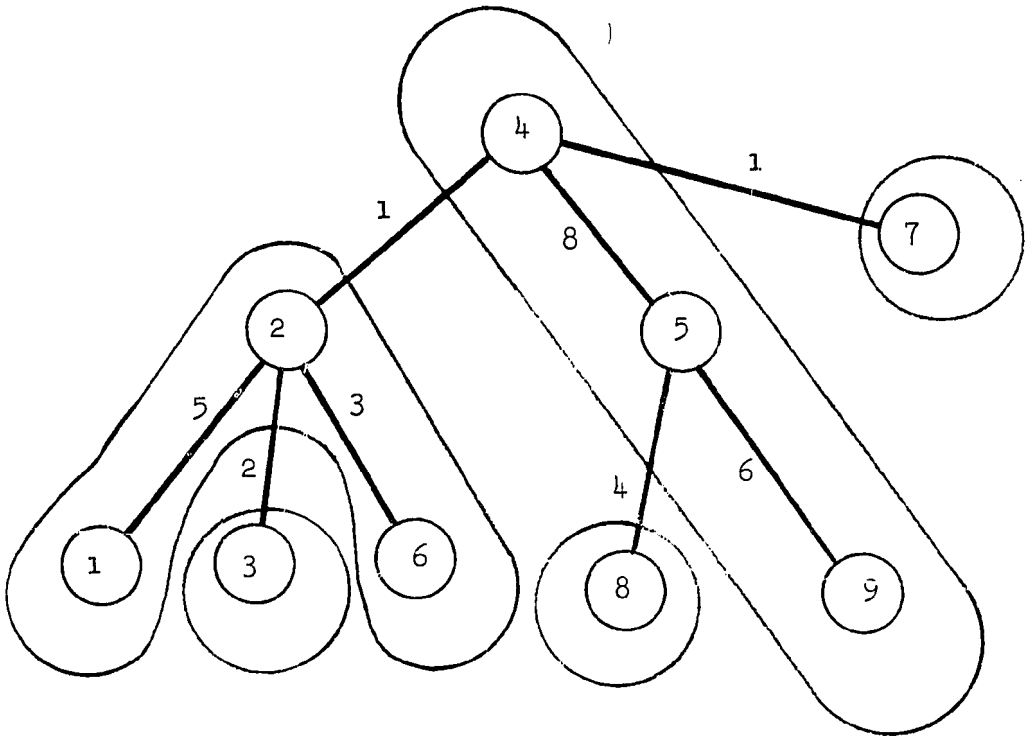


Figure 3.5 -- Partition of tree of Fig. 3.1(a)

CHAPTER IV

GRAPH LABELING

The labeling of a graph is an important part of the partitioning process since the maximum number of partitions generated on each step of this process is a function of the labeling. The time to partition an n node graph is proportional to $np_k(\log_2 p_k)$, where p_k is the number of partitions generated on step k of the partitioning algorithm. The storage requirements of the algorithm vary as np_k. Consequently, the cardinality of P_k, the set of partitions generated on step k, should be kept as small as possible for maximum efficiency.

In this chapter we show the relationship of the labeling of the graph and the size of P_k. An ad hoc labeling algorithm with a computational complexity related algebraically to the number of graph nodes is also described.

A. RELATIONSHIP BETWEEN LABELING AND SIZE OF P_k

In Section E of Chapter II we show that

$$p_k \leq x_k(f(x_k))W^{x_k},$$

where

$p_k = |P_k|$ = number of partitions generated on step k of the partitioning algorithm

$x_k = |CONN(k)_{max}| = (k-1) - i_k$

$i_k = |ISOL(k)|$

W = weight constraint

$f(x_k)$ = number of ways in which nodes in $CONN(k)_{max}$ are distributed in clusters.

In Appendix B we show that the number of operations performed on step k of the partitioning algorithm has an upper bound proportional to

$$n[p_k(\log_2 p_k)]$$

for an n node graph G. As a consequence, the sum

$$n \sum_{k=1}^{n} p_k(\log_2 p_k)$$

is directly proportional to the computation time required to partition G.

We also show in Appendix B that the growth in storage requirements for the kth step of the algorithm varies as

$$n(p_k),$$

therefore

$$n \sum_{k=1}^{n} p_k$$

can be used as a measure of the growth in the average storage requirements of the algorithm.

In order to compare the effectiveness of the many possible labelings of a given graph, we use the sum

$$S = \sum_{k=1}^{n} Z^{k-i_k},$$

where Z is a constant whose value is much greater than one. We now justify the use of this sum in measuring the effect of a given labeling upon the growth in computation time and storage for a given graph.

Since $p_k \leq x_k(x_k!)W^{x_k}$, the worst-case value of p_k grows asymptotically as Z^{x_k}, where

$$Z = \frac{cWx_k}{e},$$

e=base of natural logarithms,

$$c=[2\Pi x_k^3]^{1/(2x_k)} \doteq 1.$$

The value of $(p_k)(\log_2 p_k)$ can then be approximated by

$$Z^{x_k+1}{}_{-Z}{}^{k-i_k}$$

since

$$(p_k)(\log_2 p_k) < (Z^{x_k})(x_k)(\log_2 Z)$$

and we assume that

$$x_k(\log_2 Z) \leq Z.$$

Therefore the growth in both computation time and the average storage requirements to partition an n node graph G is proportional to

$$n\sum_{k=1}^{n} Z^{k-i_k}.$$

We use the sum above to compare different labelings of the same graph, therefore the value of n carries no added information and can be omitted. To further simplify the measure, the value of Z is assumed to be independent of k.

The objective in labeling is then to minimize

$$S = \sum_{k=1}^{n} Z^{k-i_k}$$

by maximizing for each node the size of ISOL(k),i_k.

A feature of this summation is made clearer by a change in notation:

Let $\Delta_k=i_k-i_{k-1}$ and $i_0=0$. Then

$$S = Z^{1-\Delta_1}[1 + Z^{1-\Delta_2}[1 + Z^{1-\Delta_3}[\ldots[1 + Z^{1-\Delta_n}]\ldots]].$$

Given two different labelings A and B, it is apparent that labeling A with a value of $\Delta_{i_A} > \Delta_{i_B}$ may result in a smaller value of S than labeling B

with $\Delta_{j_B} > \Delta_{j_A}$, where $j > i$. If a plot of i_k versus k is made to compare two different labelings of a graph, two situations can occur:

(1) Labeling A has a consistently higher value of i_k than labeling B (Fig. 4.1(a)). The value of S(A) for labeling A therefore is always less than that of S(B) for labeling B.

(2) The curves of i_k versus k for labeling A and labeling B cross at one or more points (Fig. 4.1(b)). The only method of comparing the two labelings is to evaluate S(A) and S(B) and choose the labeling with the smallest value.

Note that a curve of i_k versus k is monotonically nondecreasing. This is a result of the theorem of Chapter II that proves that $|ISOL(k)| \leq |ISOL(k+1)|$.

The usefulness of the global evaluation of a labeling L given by

$$S(L) = \sum_{k=1}^{n} Z^{k-i(L)_k}$$

is limited to those situations where the labeling L already exists. It is assumed here that the graph is unlabeled.

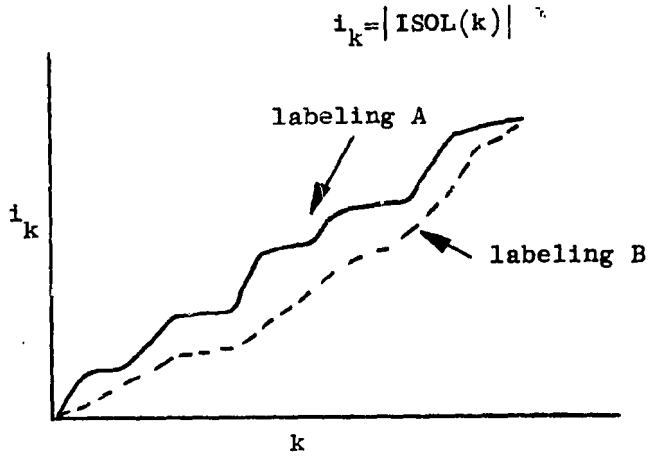
A nonenumerative algorithm that generates a globally optimal labeling has not been found. However, we now describe a locally optimal algorithm with algebraic growth in computation time.

B. LABELING ALGORITHM

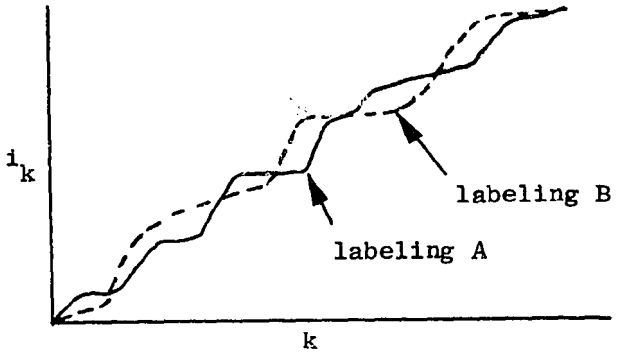
In Section A we develop the sum

$$S(L) = \sum_{j=1}^{n} Z^{j-i_j(L)}$$

to measure the labeling L of a graph G, where $i_j(L)$ is the size of ISOL(j) for labeling L. The labeling of G with the minimum value of S requires



(a) Example in which labeling A is consistently better than labeling B



(b) Example in which it is not possible to know which labeling is the best without evaluating both

Figure 4.1 -- Curves of cardinality of ISOL(k) versus k

the least storage space and computation time to partition G. Since a graph with n nodes can have n! labelings impressed upon it, a labeling procedure that consists of generating a labeling L and using S(L) to measure its effectiveness is impractical. We therefore resort to the generation of a labeling iteratively by assigning the labels in ascending order and use the increment in the partial value of S caused by the assignment of labels to unlabeled nodes to compare the effectiveness of different assignments.

If we assign labels $k_i, k_i+1, \ldots, k_j$ to a set of unlabeled nodes, the effect on the value of S can be measured by observing the term

$$\Delta S(k_i,k_j) = \sum_{j=k_i}^{k_j} Z^{j-i_j} .$$

The value of S is then given by

$$S = \Delta S(1,k_1)+\Delta S(k_1+1,k_2)+\ldots+\Delta S(k_r+1,n).$$

The basis of the labeling algorithm is to minimize the increments in the values of S, $\Delta S(k_i,k_j)$, and thereby attempt to minimize the global value of S. Note that although each value of $\Delta S(k_i,k_j)$ is minimal, the value of S may not be minimal. Section D investigates this point further. In order to find the set of unlabeled nodes whose labeling causes the minimum increment in the values of S, we develop the following rules.

The increase in the value of S is minimized by assigning labels to the nodes of the graph so that each value of i_j, the size of ISOL(j), is maximal for j=1,2,...,n. In order to become a member of the isolated of some node m, a node with label j can be adjacent to no node with label greater than or equal to m. Since nodes are assigned in ascending order, a node with label j adjacent to r unlabeled nodes can become a member of

some ISOL(m) only if the r unlabeled nodes are assigned labels less than m but greater than j. Assume that the largest label previously assigned is k; to minimize m we must then assign labels k+1,k+2,...,k+r to the r unlabeled nodes. To minimize the increment in the value of S, we then try to find that set of unlabeled nodes whose labeling allows the largest number of previously labeled nodes to become members of ISOL(m),ISOL(m+1), ...,ISOL(n) for as small a value of m as possible. To determine that set, we perform the following analysis.

Consider a step in the labeling of a graph where a number of the nodes have been labeled. Let the set of labeled nodes, each of which is adjacent to an unlabeled node, be denoted by L and the set of unlabeled nodes by U. With each labeled node i associate a subset of U, denoted by U_i, with the property that each node in U_i is unlabeled and adjacent to node i. If the largest label assigned to a node in U_i is m, then node i is a member of the isolated set of every node with label greater than m.

Some set U_j associated with a node whose label is j may be contained within U_i. For this case we define the relationship RELEASES, where i RELEASES j if:

(1) $U_j \subset U_i$

(2) $U_i=U_j$ and label i $\leq$ label j.

If the nodes in U_i are labeled such that the greatest label assigned to a node of U_i is m, then both nodes i and j are "released" to become members of the isolated set of those nodes with labels greater than m.

We then form the sets $L_{i_1}, L_{i_2}, \ldots, L_{i_r}$ of those labeled nodes adjacent to one or more unlabeled nodes using the relationship RELEASES such that

$$L_i=\{j \mid j \text{ is a node in L and i RELEASES j}\}.$$

Fig. 4.2 illustrates the definition of L_i and U_i, where $u_i=|U_i|$ and $l_i=|L_i|$.

Given the collection of sets of unlabeled nodes $U_{i_1},U_{i_2},\ldots,U_{i_r}$ and their associated sets of labeled nodes $L_{i_1},L_{i_2},\ldots,L_{i_r}$ formed by the relation RELEASES, we then select some set U_a from this collection for labeling. Let the last label assigned to a node be k. The criterion used to select the set U_a is as follows:

(1) For all pairs of sets in the collection $U_{i_1},U_{i_2},\ldots,U_{i_r}$, find the increment in the value of S caused by assigning labels $k+1,k+2,\ldots,k+u_a$ to U_a and then assigning labels $k+u_a+1$, $k+u_a+2,\ldots,k+u_a+u_b$ to U_b.

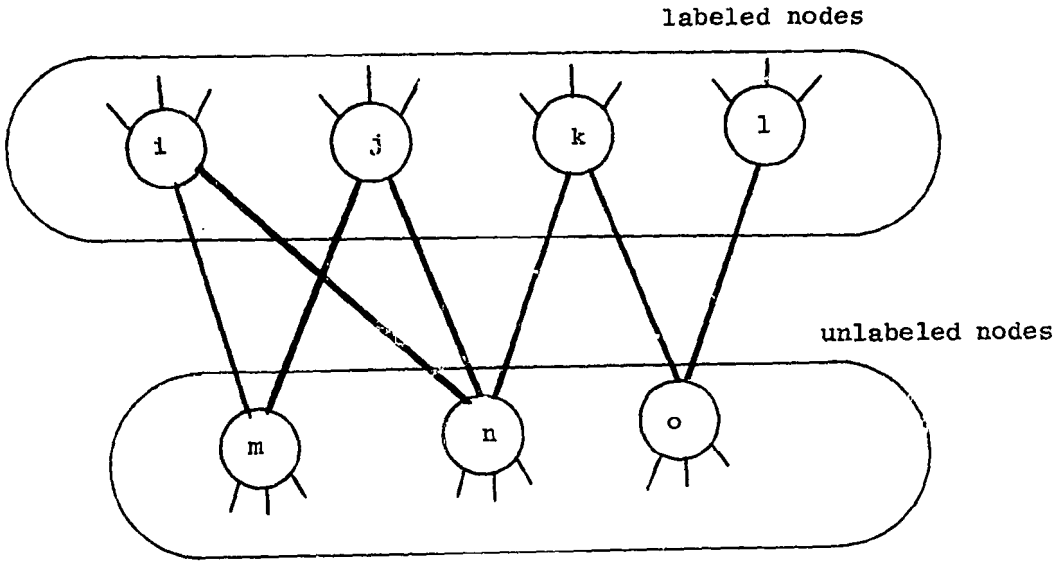
(2) Compare the results of (1) with the increase in S caused by assigning labels $k+1,k+2,\ldots,k+u_b$ to U_b and then assigning labels $k+u_b+1,k+u_b+2,\ldots,k+u_b+u_a$ to U_a.

We then label the set in the collection that causes the smallest increment in the value of S. This practice may fail to yield a value of S that is minimal because the assumptions are made that:

(1) The only sets of unlabeled nodes adjacent to labeled nodes after U_a is labeled are those in the collection U_i, where $i=i_1,i_2,\ldots,i_r$. This ignores the fact that newly created sets of unlabeled nodes adjacent to labeled nodes may be created by labeling U_a.

(2) The sets $U_{i_1},U_{i_2},\ldots,U_{i_r}$ are not modified by the labeling of U_a. This may not be true.

We have found no efficient method to detect the modification of existing sets U_i $(i=i_1,i_2,\ldots,i_r)$, or the creation of new sets $U_{i_{r+1}},U_{i_{r+2}},\ldots$, due to the labeling of U_a.



$L=\{i,j,k,l\}$ $\quad i<j<k<l$

$U=\{m,n,o\}$

$U_i=\{m,n\}$ $\quad U_j=\{m,n\}$

$U_k=\{n,o\}$ $\quad U_l=\{o\}$

Therefore:

$L_i=\{i,j\}$	$U_i=\{m,n\}$	$l_i=2$	$u_i=2$
$L_k=\{k,i\}$	$U_k=\{n,o\}$	$l_k=2$	$u_k=2$
$L_l=\{l\}$	$U_l=\{o\}$	$l_l=1$	$u_l=1$
$L_j=\{j\}$	$U_j=\{m,n\}$	$l_j=1$	$u_j=2$

Figure 4.2 -- Illustration of definitions

With these rules we calculate the effect on the value of S of labeling the nodes in some set U_a by comparing the increase in S caused by first labeling U_a and then labeling U_b, where $b=i_1,i_2,\ldots,i_r$ and $a\neq b$. If we denote the change in the value of S caused by first labeling U_a and then labeling U_b by ΔS_{ab}, the value of ΔS_{ab} is given by

$$\Delta S_{ab} = \sum_{j=k+1}^{k+u_a+u_b} Z^{j-i_j}$$

where

$$i_j = \begin{cases} i_k & k < j \leq k+u_a \\ i_k+1_a & k+u_a < j \leq k+u_a+u_b \end{cases} .$$

This increment in the value of S is then compared to that caused by first labeling U_b and then labeling U_a,

$$\Delta S_{ba} = \sum_{j=k+1}^{k+u_a+u_b} Z^{j-i_j}$$

where

$$i_j = \begin{cases} i_k & k < j \leq k+u_b \\ i_k+1_b & k+u_b < j \leq k+u_a+u_b \end{cases} .$$

The values of ΔS_{ab} and ΔS_{ba} can be simplified by forming the terms

$$\frac{\Delta S_{ab}}{Z^{k-i_k}} = \sum_{j=1}^{u_a} Z^j + \sum_{j=1}^{u_b} Z^{j+u_a-1_a}$$

$$\doteq Z^{u_a} + Z^{u_a+u_b-1_a}$$

and

$$\frac{\Delta S_{ba}}{Z^{k-i_k}} \doteq Z^{u_b} + Z^{u_a+u_b-1_b} .$$

The labeling of U_a, and then U_b, then results in a smaller increase in the value of S than the labeling of U_b, and then U_a, if

$$Z^{u_a} + Z^{u_a+u_b-l_a} < Z^{u_b} + Z^{u_a+u_b-l_b}.$$

Let

$$Z^{u_a} + Z^{u_b+u_a-l_a} + K_{ab} = Z^{u_b} + Z^{u_a+u_b-l_b}.$$

Then

$$K_{ab} = Z^{u_a+u_b}\left[\frac{1}{Z^{u_a}} - \frac{1}{Z^{l_a}} - \left(\frac{1}{Z^{u_b}} - \frac{1}{Z^{l_b}}\right)\right]$$

and the labeling of U_a and then U_b results in a lower increment in S than the labeling of U_b and then U_a if $K_{ab} > 0$.

In general, given the sets $U_{i_1}, U_{i_2}, \ldots, U_{i_r}$ and $L_{i_1}, L_{i_2}, \ldots, L_{i_r}$, we use the following procedure to find that set U_i such that $K_{ij} \geq 0$ for $j=i_1, i_2, \ldots, i_r$ and $j \neq i$.

(1) For $k=i_1, i_2, \ldots, i_r$ form the differences $u_k - l_k$ and separate the sets U_k into three disjoint sets:

(a) Set I = $\{U_k | u_k - l_k < 0\}$

(b) Set II = $\{U_k | u_k - l_k = 0\}$

(c) Set III = $\{U_k | u_k - l_k > 0\}$.

(2) If sets I and II are vacuous, then label the element of set III, U_i, such that l_i is maximum. If two or more elements of set III satisfy this criterion, then label the element of set III satisfying the criteria of maximum l_i and minimum u_i.

(3) If set I is empty, then $K_{ab}=K_{ba}$. Use criterion listed in (5) below.

(4) If set I is not empty, then choose that element of set I such that u_i is minimum. If several elements of set I have minimal values, select that element with minimum u_i and maximum l_i.

(5) If two or more elements are equivalent from the standpoint of the above tests, then label that set U_i sharing the most elements with the other sets U_k for $k \neq i$. This practice minimizes the number of unlabeled nodes in the graph adjacent to labeled nodes, allowing the labeled nodes to become nodes in the isolated set ISOL(j) for lower labels j.

To initiate the labeling process we select some node as the node with label 1. A candidate for label 1 can be selected by:

(1) choosing that node adjacent to the least number of nodes in the graph;

(2) letting each node of the graph be node 1 in turn and performing the labeling algorithm n times for an n node graph.

The first alternative is based upon the fact that labeling the node with the least number of adjacent nodes allows node 1 to become a node in some ISOL(j) where j is minimal.

The labeling algorithm is summarized in the flow chart of Fig. 4.3.

The computational growth rate of this algorithm is summarized in Table 4.1. This table shows that the growth in computation time varies as n^3, where n is the number of graph nodes.

An implementation of the partitioning of a graph with cutpoints is described in Appendix C. This implementation reduces the growth in computation for the labeling process to $c(n')^3$, where n' is the number of nodes in the largest block of the graph and c is a constant.

C. EXAMPLE

We present an example of the labeling algorithm for the graph of Fig. 4.4(a):

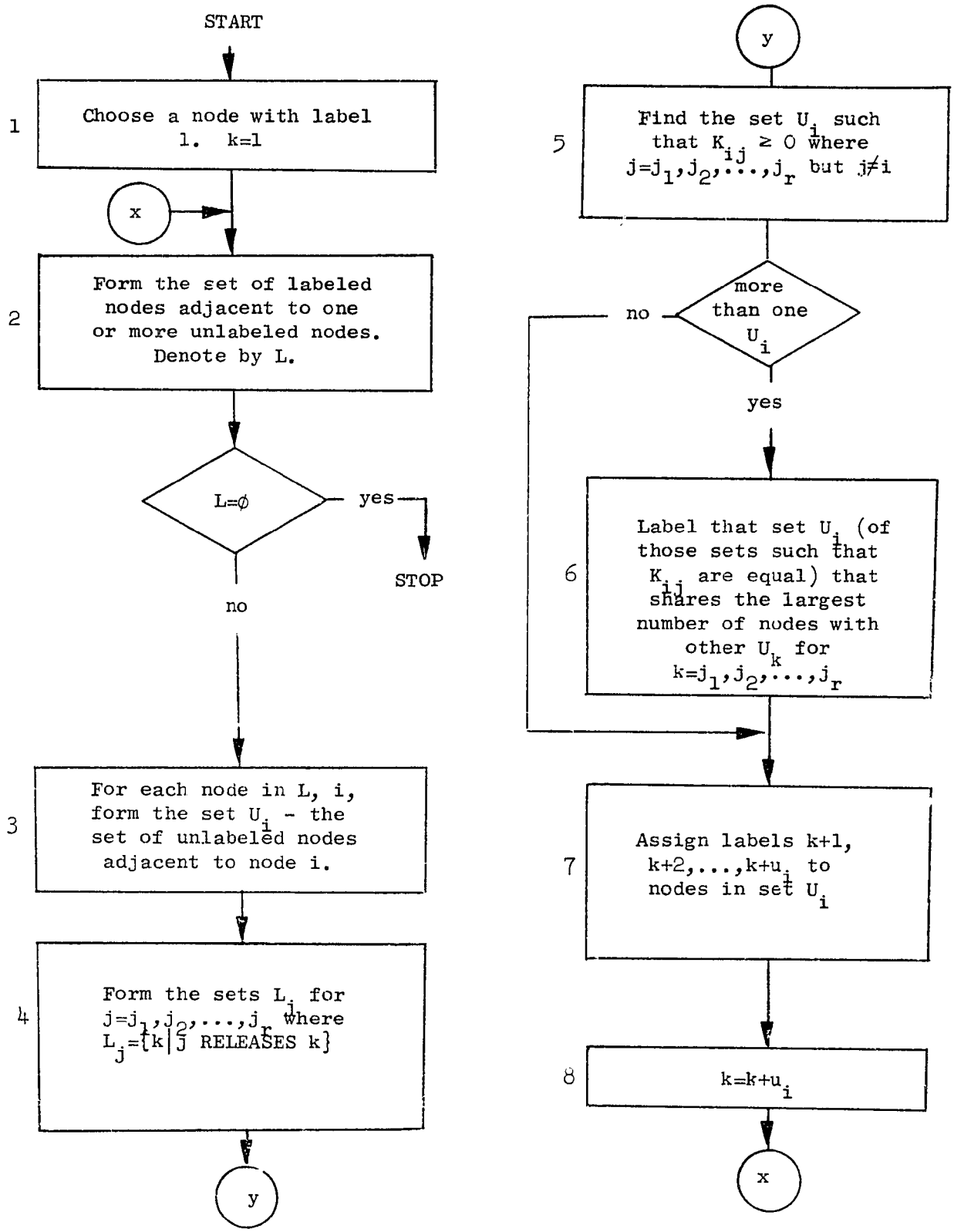


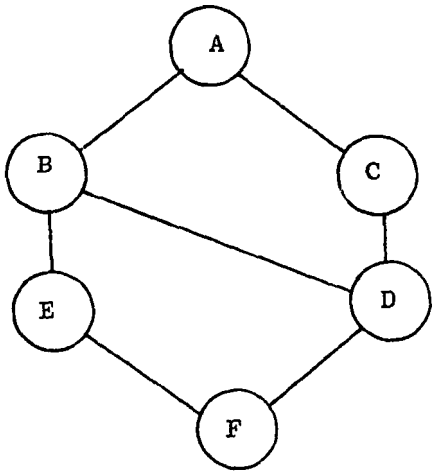
Figure 4.3 -- Flowchart of labeling algorithm

Table 4.1 -- Operations required to label an n node graph

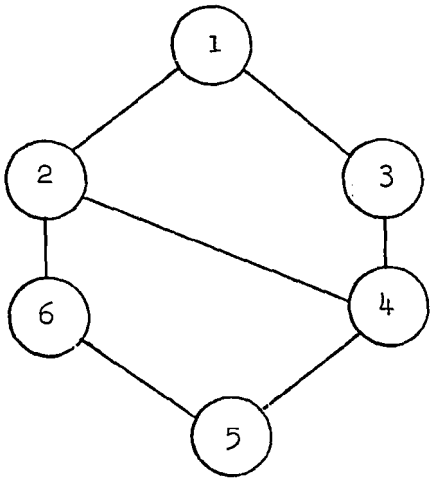
STEP*	OPERATIONS/STEP**	TOTAL OPERATIONS
1	$c_1 n^2$	$c_1 n^2$
2	$c_2 n$	$c_2 n^2$
3	$c_3 n^2$	$c_3 n^3$
4	$c_4 n^2$	$c_4 n^3$
5	$c_5 n^2$	$c_5 n^3$
6	$c_5 n^2$	$c_6 n^3$
7	$c_7 n$	$c_7 n^2$
8	c_8	$c_8 n$

* Refer to Fig. 4.3 for step number

** The constants $c_1, c_2, \ldots, c_8$ are dependent upon the particular implementation used for the algorithm



(a) Given graph



k	ISOL(k)
1	∅
2	∅
3	∅
4	{1}
5	{1,3}
6	{1,3,4}

(b) Resulting labeling

Figure 4.4 -- Example of labeling process

(1) Since nodes A,C,E, and F each have the minimum number of adjacent nodes, select A arbitrarily as the node with label 1.

(2) $L = \{1\}$ $U=\{B,C,D,E,F\}$

Thus

$U_1=\{B,C\}$ subsets of L: $L_1=\{1\}$

$U_1=\{B,C\}$

Let node B have label 2 and node C label 3.

(3) $L=\{2,3\}$ $U=\{E,D,F\}$

Thus

$U_2=\{E,D\}$ subsets of L: $L_2=\{2,3\}$

$U_3=\{D\}$ $U_2=\{E,D\}$

$L_3=\{3\}$

$U_3=\{D\}$

$K_{23}=K_{32}$ therefore let D have label 4.

(4) $L=\{2,4\}$ $U=\{E,F\}$

Thus

$U_2=\{E\}$ subsets of L: $L_2=\{2\}$

$U_4=\{F\}$ $U_2=\{E\}$

$L_4=\{4\}$

$U_4=\{F\}$

$K_{24}=K_{42}$ therefore node F has label 5.

(5) Node E has label 6 since it is the only node left.

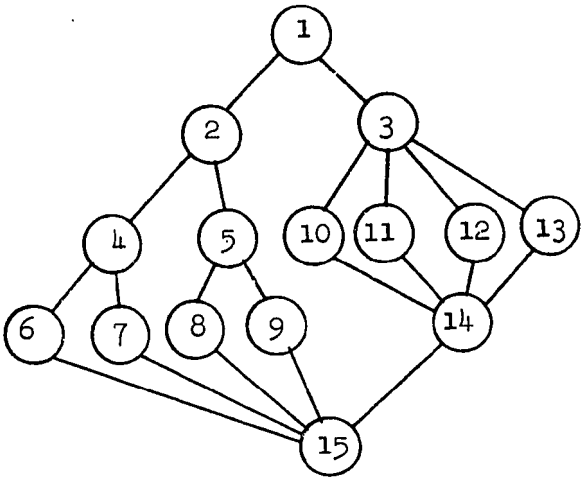
Fig. 4.4(b) shows the resulting labeled graph.

D. COMMENTS ON THE OPTIMALITY OF THE LABELING ALGORITHM

A counterexample of the optimality of the algorithm above is shown in Fig. 4.5. Fig. 4.5(a) shows the labeling that results from the labeling algorithm and Fig. 4.5(b) another labeling that deliberately defies the criterion employed in the algorithm. A comparison of the values of S for each labeling shows that the second labeling results in a lower value.

This phenomenon occurs because the labeling algorithm only examines local data. One method of partially overcoming this problem is the simple "lookahead" strategy now described.

Given a situation in which L has been separated into subsets $L_1, L_2, \ldots, L_x$, allow the nodes in each subset L_i to become isolated nodes by labeling the set U_i. Then, perform the local labeling algorithm for a few steps. The set U_k resulting in the best overall value of S is then labeled first. This practice avoids the problem of Fig. 4.5 while increasing the computation time moderately.

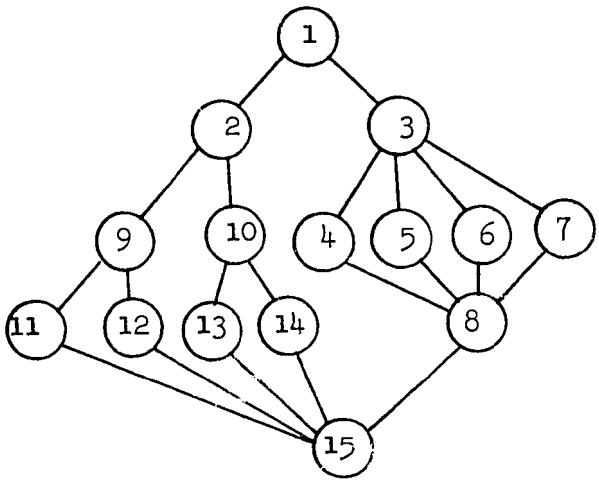


$\Delta(k)=ISOL(k)-ISOL(k-1)$

k	$\Delta(k)$
4	1
6	2
8	4
10	5
14	3
15	10,11,12,13

$S \doteq 2Z^9$

(a) Labeling produced by labeling algorithm



k	$\Delta(k)$
4	1
8	3
9	4,5,6,7
11	2
13	9
15	10

$S \doteq 4Z^6$

(b) Labeling that defies labeling criterion used in labeling algorithm

Figure 4.5 -- Counterexample to local labeling criterion

CHAPTER V

CONCLUSIONS

A. SUMMARY OF RESULTS

In this thesis we investigate the problem of partitioning the integer weighted nodes of a graph into clusters so that the values of the edges cut are minimized.

Chapter II describes a dynamic programming procedure that generates "feasible" partitions of an n node graph G. A partition is feasible if each of its clusters satisfies the following restrictions:

(1) The sum of the node weights in the cluster is equal to or less than a given weight constraint.

(2) The cluster nodes form a connected subgraph of G.

The nodes of G are first assigned unique labels 1,2,...,n. Then stage k of the dynamic programming procedure generates feasible partitions of those nodes with labels $\leq$ k.

The number of feasible partitions for a cyclic k node graph grows exponentially in k. Since the growth in computation time is proportional to

$$n \sum_{k=1}^{n} p_k(\log_2 p_k)$$

where p_k is the number of partitions generated on the kth step of the partitioning process, the use of the dynamic programming procedure to generate all feasible partitions is quite inefficient.

We then introduce the concept of the isolated set. This concept is based upon the connectivity requirements of each cluster of a feasible partition and limits the number of partitions generated on the kth step of

the dynamic programming procedure to less than

$$x_k f(x_k) W^{x_k} \quad \text{where } 1 \le f(x_k) < x_k! \ .$$

Here x_k is the number of nodes of the graph with labels less than k that can be clustered with node k. If x_k is much less than k for k=1,2,...,n and W<<n, the number of partitions that must be generated by the dynamic programming procedure is substantially less than the number of feasible partitions.

A graph with cutpoints can be partitioned by first partitioning the blocks of the graph, then combining these partitions to generate an optimal partition of the entire graph. The maximum number of partitions generated on a step of the partitioning process is a function of the number of nodes in a block and not the graph itself.

In Chapter III the results of Chapter II are applied to the partitioning of a tree. The special properties of the tree result in an algorithm whose computation time and storage requirements grow linearly with the number of graph nodes.

A basic requirement of the partitioning algorithm is the assignment of a unique integer label to each node. In Chapter IV we show the relationship between the labeling impressed upon the graph and the growth in the number of partitions generated on each step of the partitioning algorithm. An ad hoc labeling algorithm is also described.

B. FUTURE RESEARCH

We have investigated the problem of partitioning a connected graph G into disjoint clusters with the objective of minimizing the value of the edges cut by the partition. A logical extension of the partitioning problem is the investigation of the problem of finding a minimum-valued

cover of G. A cover differs from a partition in that a cover is the distribution of the nodes of G into clusters $\{c_i\}$ (i=1,2,...,k) where $c_i \cap c_j$ need not be empty. A cover can often result in a lower value of the intercluster edges than a partition of G [Kernighan, 1969].

An algorithm that solves the covering problem can be used to cluster logic gates onto integrated circuit modules. The objective here is to minimize the number of intermodule connections at the expense of duplicating gates. This problem is discussed extensively in the article by Oden, Russo, and Wolff [1971].

The labeling algorithm developed in this thesis is locally optimal. Therefore, an investigation of algorithms to efficiently generate a globally optimal labeling is warranted.

The tree partitioning algorithm, because of its efficiency, can form the basis of a partitioning algorithm for cyclic graphs. This algorithm can efficiently generate a partition whose value may not be optimal but is within a given bound.

APPENDIX A

AN ANALYSIS OF THE LOWER BOUND ON THE NUMBER OF FEASIBLE PARTITIONS FOR A CONNECTED k NODE GRAPH

A. LOWER BOUND ON NUMBER OF FEASIBLE PARTITIONS IGNORING WEIGHT CONSTRAINT

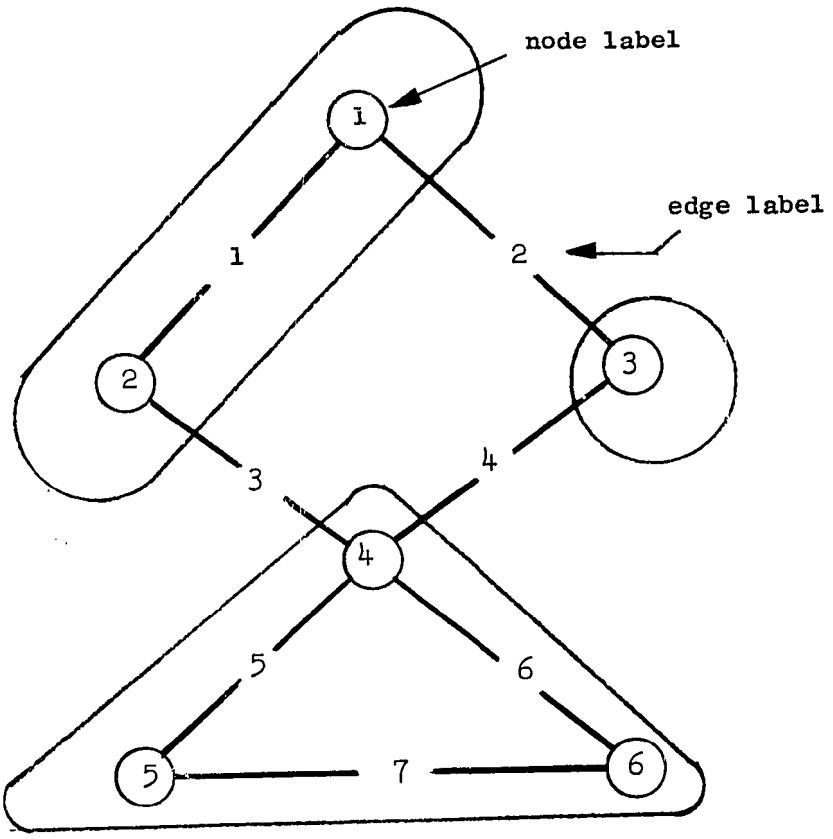
Let the e edges of a graph G be assigned unique integer labels 1,2,...,e. A binary variable e_k can then be associated with the edge with label k and represents the condition of edge k in a partition of G -- i.e.:

$$e_k = \begin{cases} 1 & \text{if edge k is within a cluster of the partition} \\ 0 & \text{if edge k is cut by the partition.} \end{cases}$$

We can then represent a partition of G by a binary sequence of length e, where the kth bit of the sequence represents the condition of edge k. This representation is unique, as we now show.

Assume that two binary sequences S_1 and S_2 represent the same partition of G. If S_1 is not equal to S_2, there must be at least one bit, e_i, that is a 1 in one sequence and a 0 in the other. This situation cannot possibly occur since an edge cannot be both cut and contained in a cluster of the same partition, thus $S_1=S_2$.

A graph with k nodes has from k-1 to k(k-1)/2 edges. If each binary sequence represents a partition, a graph with e edges has no more than 2^e feasible partitions. There are not, however, 2^e feasible partitions of an e edge cyclic graph because certain combinations of bits in an e bit binary sequence represent no partition of G. An illustration of this fact is given in Fig. A.1. In particular, let an arbitrary labeling be impressed upon the edges of the graph G. Assume that a cycle of G, whose



partition: (1,2)(3)(4,5,6)

valid representation: 1000111

invalid representation: 1000110

Figure A.1 -- Illustration of an invalid partition representation

length is c, contains the set of edges with labels $\{i_1, i_2, \ldots, i_c\}$. Any e bit sequence representing a partition of G cannot contain a combination of the bits $e_{i_1}, e_{i_2}, \ldots, e_{i_c}$, in which one of these bits is a zero and the rest are ones. Any other combination of the bits $e_{i_1}, e_{i_2}, \ldots, e_{i_c}$ is valid if no cycle is contained within this cycle.

The lower bound on the number of feasible partitions of a k node cyclic graph is 2^{k-1}, as we now show.

Let a spanning tree st(G) of a graph G be formed. Let G have k nodes, e edges, and let G be connected and cyclic. There are k-1 edges and no cycles in st(G), therefore st(G) has 2^{k-1} feasible partitions. Each feasible partition of st(G) is also a feasible partition of G. This follows from the fact that a feasible partition of st(G) can always be transformed to a feasible partition of G by adding edge (i,j) to a cluster of a partition of st(G) if nodes i and j are in the same cluster and edge (i,j) is contained in G, but not st(G). Note that this is true regardless of the weight constraint. Also, there are feasible partitions of G that are not feasible partitions of st(G). For example, if edge (i,j) is an edge of G not in st(G), then a partition with a cluster containing nodes i and j alone is feasible for G but not for st(G).

If P_k denotes the set of feasible partitions for a k node connected graph, then

$$|P_k|_{\max} = 2^{k-1} \text{ for the tree}$$

and

$$2^{k-1} < |P_k|_{\max} < 2^{(k-1)(k/2)} \text{ for a cyclic graph.}$$

B. LOWER BOUND ON NUMBER OF FEASIBLE PARTITIONS FOR A WEIGHT CONSTRAINT OF W

We have shown that a k node tree has the fewest feasible partitions of any connected k node graph, ignoring the weight constraint. Since the feasible partitions of the spanning tree of a graph G are a subset of the partitions of G, we can find a lower bound on the feasible partitions of a connected graph by analyzing the two tree types shown in Fig. A.2.

These trees represent the two extremes in spanning trees of a k node graph. The tree of Fig. A.2(a) has the fewest levels of any k node tree and each connected set contains the same node. The tree of Fig. A.2(b) has the maximum number of levels for a k node tree, and each connected set is one element contained in no other connected set.

The number of feasible partitions of the tree of Fig. A.2(a) is given by the summation

$$|P_k| = \sum_{i=1}^{W-1} \binom{k}{i} .$$

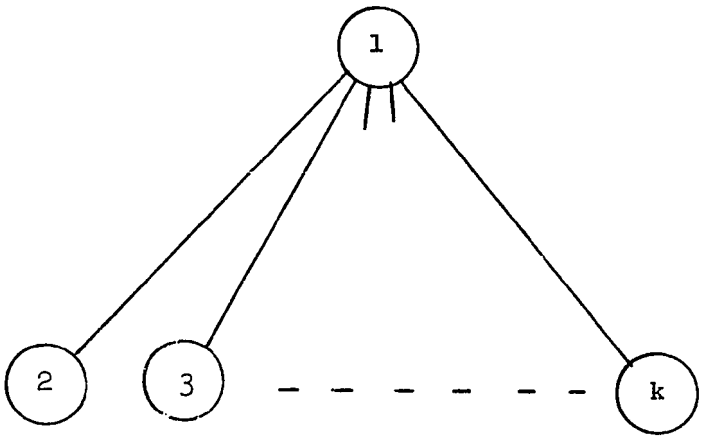
We derive this number by noting that no cluster of a feasible partition can contain more than one node unless node 1 is in that cluster. A lower bound for this summation is given by

$$\binom{k}{W-1} = \frac{k!}{(k-W+1)!\,(W-1)!} \qquad \text{for } W-1 \le k/2.$$

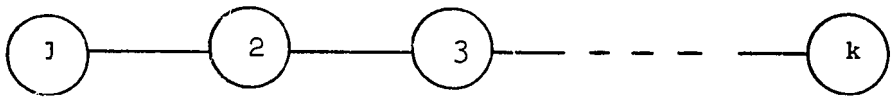
Letting $x = \frac{W-1}{k}$, and using Stirling's approximation,

$$|P_k| > \left[\frac{\left(\frac{1-x}{x}\right)^{x}}{(1-x)} \right]^{k} .$$

Letting the term



(a) Minimum-level tree



(b) Maximum-level tree

Figure A.2 -- Minimum- and maximum-level spanning trees

$$\left[\frac{\frac{1-x}{x}}{(1-x)} \right]^{x} = f(x) ,$$

we can show that $1 < f(x) < 2$ for $0 < x < \frac{1}{2}$.

A recurrence relationship for the number of feasible partitions of the tree of Fig. A.2(b) can be derived from the fact that no more than W nodes, hence W-1 edges, can appear together in a single cluster for a weight constraint of W. Consequently, a binary sequence representing a feasible partition of this tree cannot have a consecutive sequence on W or more ones.

The number of binary sequences representing feasible partitions for a weight constraint W is given by

$$|P_k| = 2^{k-1} - b_{k-1} ,$$

where

$$b_r = 2b_{r-1} + 2^{r-W-1} - b_{r-W-1}$$

with initial conditions

$$b_1=b_2=\ldots=b_{W-1}=0 \text{ and } b_W=1.$$

The solution to this recurrence relationship for W=2 is

$$|P_k| = F_k ,$$

where F_k is the kth Fibonacci number. The following theorem proves that this is the minimum number of partitions for a nontrivial weight constraint.

Theorem

If B(k,W) denotes the set of k bit binary sequences with no subsequence of W or more adjacent ones, and if $d_k(W)=|B(k,W)|$, then

$$d_k(x) < d_k(y) \quad \text{if } x < y.$$

Proof

$B(k,W) \subseteq B(k,W+1)$ since all sequences in $B(k,W)$ are sequences in $B(k,W+1)$. $B(k,W+1)$ properly contains $B(k,W)$ since a k bit sequence with a subsequence of W adjacent ones is a sequence in $B(k,W+1)$ and not in $B(k,W)$; yet, all sequences in $B(k,W)$ are sequences in $B(k,W+1)$. Since $B(k,W) \subset B(k,W+1)$, $d_k(W) < d_k(W+1) < d_k(W+2)\ldots$, thus $d_k(x) < d_k(y)$ if $x < y$.

The sequences in the set $B(k,W)$ are in one-to-one correspondence to the partitions in P_{k+1} for the simple tree of Fig. A.2(b). Therefore, the number of partitions for this tree increases from F_k for W=2 to 2^{k-1} for $W \geq k$. ■

APPENDIX B

IMPLEMENTATION OF BASIC PARTITIONING ALGORITHM

Much of the analysis of Chapters II, III, and IV is based upon the assumption that the computational and storage requirements associated with some step of the partitioning algorithm are directly proportional to

$$n[p_k * \log_2 p_k],$$

where n is the number of graph nodes and p_k is the number of partitions generated on step k. To support this assumption, an implementation of the basic partitioning algorithm outlined in Fig. 2.11 is now described.

A. DATA STRUCTURE

Let P_k be the set of partitions associated with step k of the partitioning algorithm. The information associated with each partition in the set of partitions P_k is shown in Fig. B.1. The fields of the two data types are summarized below:

(1) HEADER (one per partition)

- (a) DFLAG: a flag used to signal the existence of a partition to be deleted.
- (b) VAL: value of partition

(2) BODY (one entry per graph node)

- (a) CC-FLAG: Warns that the cluster containing this node has been used previously to form a new partition.
- (b) REP: The n REP entries form a unique representation of a partition. This representation is used in conjunction with the WT entries to detect the dominant partitions of each step.

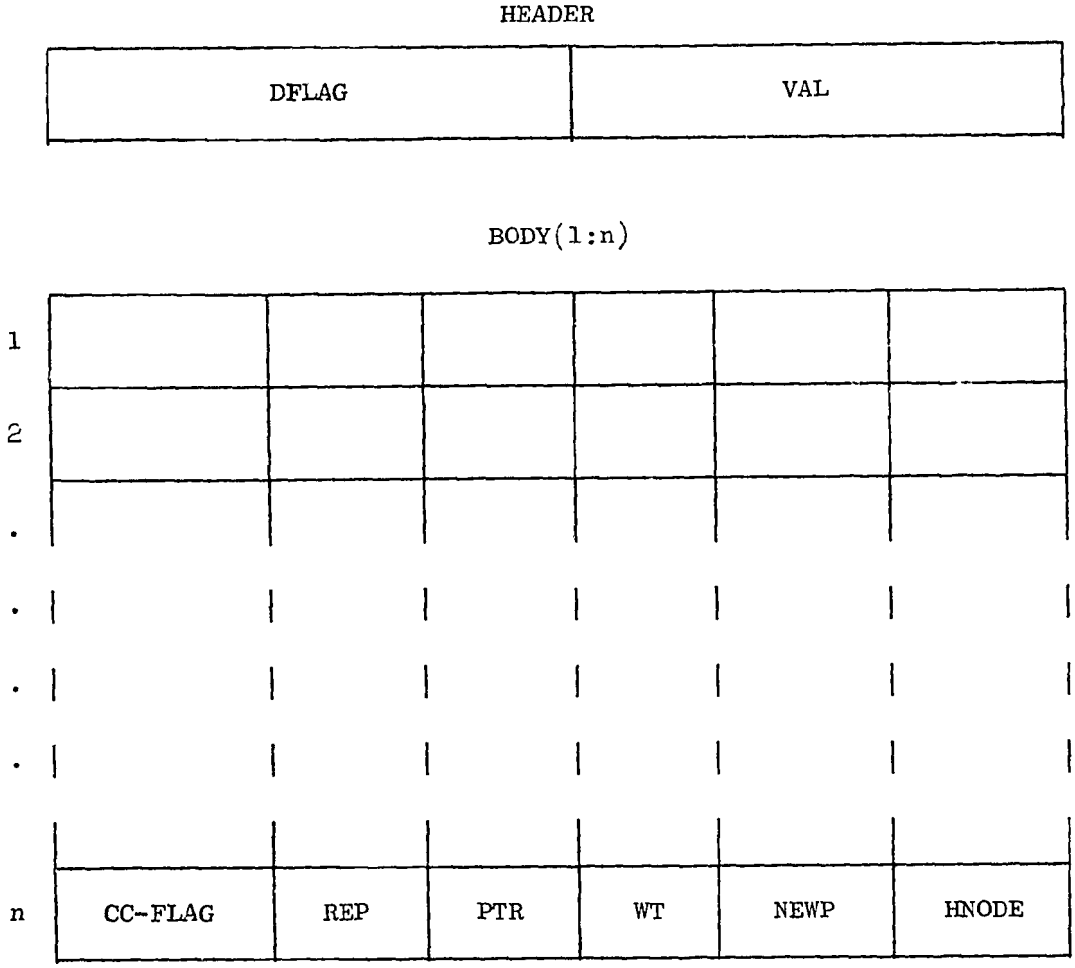


Figure B.1 -- Data structure for a partition

(c) PTR: Used to link the BODY elements of nodes in the same cluster.

(d) WT: Weight associated with the cluster in which this node exists.

(e) NEWP: Contains pointer to a partition previously created from this partition. This entry allows the value of a partition with a cluster that has two or more elements of CONN(k) to be updated correctly.

(f) HNODE: Contains the highest-numbered node on the path from node j to node k, where $j \in$ CONN(k) and the path is that path that results in j being an element of CONN(k). The purpose of this entry is to handle situations where a cluster violates the connectivity constraint locally, but does not do so globally. When HNODE $<$ k, the unconnected cluster containing k and j never becomes connected by the addition of a node with label $>$ k. Consequently, the partition can be deleted.

Each partition has a unique representation consisting of the n-tuple formed by the REP entries of the BODY data associated with that partition. This representation is used to detect the dominant partitions of P_k. The Isolated Set Theorem proves that all but the dominant partitions can be deleted from P_k before performing step k+1.

A cluster of a partition is uniquely identified by assigning the same integer i to each REP entry associated with the non-isolated nodes in that cluster. The integer i is the label of that node in the cluster that

becomes a member of some $\Delta(j)=ISOL(j)-ISOL(j-1)$ for a maximum value of j. If two or more nodes in a cluster belong to the same $\Delta(j)$, then choose as identifier that node with the smallest label. The reason for selecting this form of identification for a cluster is explained below.

The REP entries associated with isolated nodes are zeroed. We then use the representation, as well as the weights of those clusters containing a node in $CONN(k)_{max}$, to find similar partitions. Two partitions are similar if:

(1) their representations are equal,

(2) the weight of a cluster with identifier i, where $i > 0$, of the first partition equals that of cluster i of the second partition.

We use the node r that is an element of $\Delta(j)$ for the largest label j to identify the cluster in which it exists because this avoids changing the cluster identification until a new node is added to the cluster. A cluster's identification changes for the following reasons:

(1) All nodes in the cluster become isolated nodes, in which case all REP entries become zero.

(2) The cluster is modified by the inclusion of node k, in which case the cluster identification is updated if required.

The use of the partition representation is illustrated in Fig. B.2.

B. ALGORITHM

(1) Label graph.

(2) Find CONN(k) and ISOL(k) for all nodes k. Also create a matrix HI(j,k), where $j \in CONN(k)$ and j is not adjacent to k. Here the entry in the matrix is the largest label of a node on the path from

	REP	PTR	WT
1	0	2	0
2	2	1	6
3	0	3	1
4	0	5	0
5	5	4	11
6	0	6	0
7	0	7	0
8	0	8	0

ISOL(5)={1,3,4}

$2 \in \Delta(7)$

$5 \in \Delta(8)$

(a) Partition (1,2)(3)(4,5)

$r_{i,6}$=(1,2)(3)(4,5,6)

$p_{j,6}$=(1)(2)(4)(3,5,6)

ISOL(6)={1,3,2,4}

$p_{i,6}$

	REP	PTR	WT
1	0	2	0
2	0	1	2
3	0	3	1
4	0	5	0
5	6	6	0
6	6	4	3

$p_{j,6}$

	REP	PTR	WT
1	0	1	1
2	0	2	1
3	0	5	0
4	0	4	1
5	6	6	0
6	6	3	3

(b) Equivalent partitions

Figure B.2 -- Examples of partition representations

node j to node k that results in node j being in the set CONN(k). Also, find $\Delta(k)$=ISOL(k) - ISOL(k-1), where ISOL(0)=ϕ.

(3) P_1={(1)}, k=1.

(4) k=k+1. If k > n, then exit.

(5) If CONN(k) is empty, go to (7). Else, select a node in CONN(k) -- let it be j -- and delete j from CONN(k).

(6) Form new partitions:

(a) For i=1 to $|P_{k-1}|$ let $q=p_{i,k-1}$ -- i.e. create a new partition of P_k whose data is initialized to the contents of $p_{i,k-1}$, the ith partition in P_{k-1}.

(b) Let T1=q.REP(j).

(c) If q.CC-FLAG(T1)=1, then cluster T1 has been previously modified by the inclusion of some other node in CONN(k). Update the value of the partition previously created by adding the value of edge(j,k) to VAL of the partition pointed to by q.NEWP(T1). Go to (6a).

(d) If q.WT(T1)+WEIGHT(k) > W, then go to (6a). Else, q.WT(T1)= q.WT(T1)+WEIGHT(k).

(e) q.VAL=q.VAL+VALUE[edge(j,k)].

(f) If q.HNODE(T1)=k, then $p_{i,k-1}$.DFLAG=1.

(g) If VALUE[edge(j,k)]=0, then enter HI(j,k) in q.HNODE(T1).

(h) If T1 is a node in $\Delta(a)$ and k is a node in $\Delta(b)$, then replace all REP entries whose identifier is T1 with k if a < b and set q.REP(k)=k. Else, q.REP(k)=T1.

(i) T2=PTR(j), PTR(j)=k, PTR(k)=T2.

(j) $p_{i,k-1}$.NEWP(T1)=pointer to storage space associated with q.

(k) Go to (6a).

(7) If $\Delta(k+1)$ is empty, go to (9). Else, select a node in $\Delta(k+1)$ -- let it be x -- and delete x from $\Delta(k+1)$.

(8) Zero all REP(x) entries associated with the partitions in P_{k-1} and P_k. Go to (7).

(9) Create 1-adjacencies of P_k by taking each partition in P_{k-1} whose DFLAG entry is zero and entering k in REP(k). This is equivalent to creating a partition with node k clustered alone. Delete all partitions in P_{k-1}.

(10) Determine those partitions of P_k that are dominant and delete all other partitions. We can implement this step by using the concept of the AVL tree [Adel'son-Vel'skii, Landis, 1962] [Foster, 1965]. An AVL tree is defined as follows:

For every node of an AVL tree, the length of the longest path in the left subtree differs from the length of the longest path in the right subtree by no more than one branch. Fig. B.3 illustrates an AVL tree.

A full description of the data structures associated with AVL trees, and searching and inserting data using AVL trees, is given in Stone [1972].

An AVL tree has the advantage that the asymptotic growth to search or insert data into the tree grows as $\log_2(r)$, where r is the number of nodes in the tree. The maximum number of nodes in the AVL tree used to find the dominant partitions of P_k is $p_k=|P_k|$. We can therefore find the dominant partitions of P_k in a number of operations whose upper bound is proportional to $p_k(\log_2 p_k)$ times the number of operations associated with the comparison of two partitions.

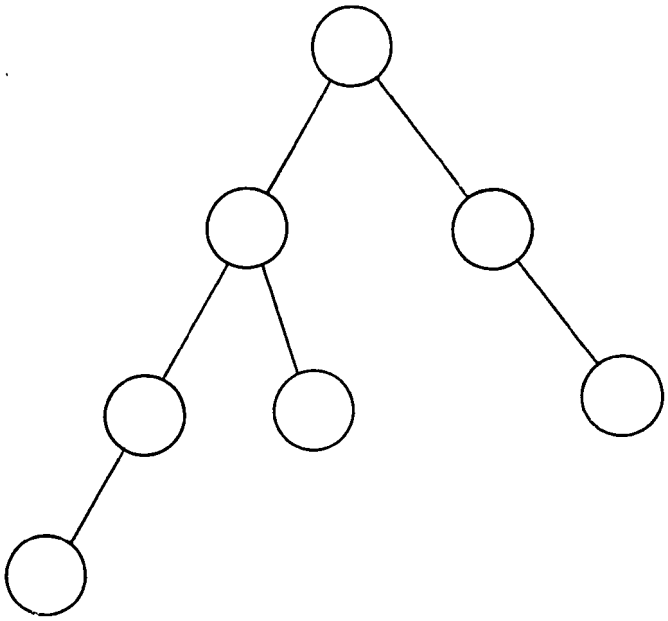


Figure B.3 -- An example of an AVL tree

We now describe how the AVL tree is used to find the dominant partitions of P_k.

In Section D of Chapter II we define two partitions as <u>similar partitions</u> if:

(1) they have equal distributions of the nodes in CONN(k) in their clusters;

(2) clusters containing the same subset of CONN(k) have the same weight in both partitions.

When translated into the data structure of Fig. B.1, two partitions p and q are similar if:

(1) they have equal representations, i.e. p.REP(i)=q.REP(i) for i=1,2,...,n;

(2) p.WT(j)=q.WT(j) where j is a nonzero cluster identifier.

The partition p <u>dominates</u> partition q if:

(1) p and q are similar;

(2) p.VAL $\geq$ q.VAL*.

To find the dominant partitions of P_k, we select some partitions p in P_k that has not yet been inserted in the AVL tree and search the tree for a similar partition. If a partition similar to p is found, then the partition of greater value is left in the tree, and the partition of lesser value is deleted from P_k. If two similar partitions have equal value, then the partition being inserted, p, is deleted from P_k. If no partition in the AVL tree is similar to p, then p is inserted in the tree.

* If two partitions have equal values, one is arbitrarily chosen as the dominant partition.

The data associated with each node of the tree consists of the n REP entries associated with a partition and a maximum of n WT entries, one for each cluster containing a node in $CONN(k)_{max}$. A comparison of two partitions then takes a number of operations proportional to n for an n node graph.

(11) Go to (4).

C. GROWTH RATE

A summary of the number of operations required to perform steps 4 through 11 of the algorithm (these steps are performed once per partition) is given in Table B.1. We see that step 10 dominates the growth in computation time. Therefore the worst-case growth in computation time varies asymptotically as

$$n[p_k(\log_2 p_k)]$$

where p_k equals the number of partitions generated on step k and n equals the number of nodes in the graph. Stone [1972] shows that the number of words of storage required to use an AVL tree grows as the number of nodes in the tree; there are no more than p_k nodes in the tree. The data outlined in Fig. B.1, however, is proportional to $n(p_k)$, consequently the storage requirements of the kth step of the algorithm grow as

$$n(p_k).$$

Table B.1 -- Number of operations required to form partitions generated on step k of the partitioning algorithm.

STEP	OPERATIONS
4,5,6,11	$(c_1+c_2W)p_k$ *
7,8	$(k+1)c_3p_k$ **
9	$c_4(p_{k-1})$
10	$c_5(np_k)(\log_2 p_k)$

Notes:

* Step 6_h may require up to c_2W operations for each partition, where W is the weight constraint.

** $|\Delta(k+1)|<k$

p_k=number of partitions generated on step k of algorithm

n= number of graph nodes

c_1, c_2, c_3, c_4, c_5 are constants dependent upon implementation of operations.

APPENDIX C

AN IMPLEMENTATION OF THE GRAPH PARTITIONING PROCESS FOR A GRAPH WITH CUTPOINTS

The Block Independence Theorem proves that an optimal partition of a graph G with one or more cutpoints can be created by generating the partitions of each block of G independently, and combining these partitions.

The only nodes of a block B contained in other blocks of G are cutpoints. This means that there can be a maximum of $x(x!)W^x$ partitions of a block with x cutpoints since all other nodes in the block are "isolated" nodes, i.e. are adjacent to no node in another block. If some block of G has x_1 cutpoints and another block has x_2, then the process of combining the partitions of these blocks may take up to $[x_1(x_1W/e)^{x_1}][x_2(x_2W/e)^{x_2}]$ steps.

We derive a partitioning algorithm here whose computation time grows asymptotically as

$$n(\bar{n}^2)(p \log_2 p)$$

and whose storage requirements grow as

$$Wp.$$

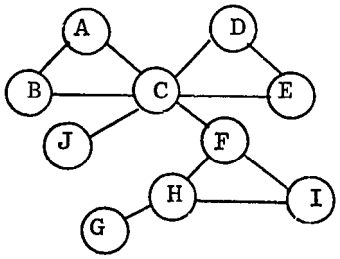
Here W is the weight constraint, n is the number of nodes in the graph G, $\bar{n}$ is the number of nodes in the largest block of G, and p is the largest number of partitions generated in partitioning a block of G. If the blocks of G have substantially fewer nodes than G, a large reduction in computation time and storage space is possible.

A. PARTITIONING ALGORITHM

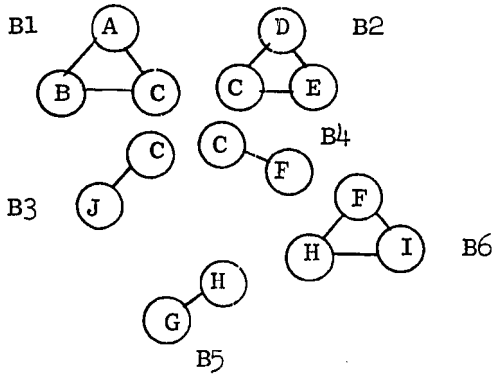
If a connected graph G has a nonvoid set of cutpoints $\{c_j\}$ and an associated set of blocks $\{B_i\}$, the block-cutpoint graph of G, denoted by bc(G), is a tree with node set $V=\{B_i\} \cup \{c_j\}$ [Harary, 1969]. Here a node B_i is associated with block B_i and a node c_j with cutpoint c_j. Two nodes are adjacent if one node corresponds to a block B_i and the other to a cutpoint c_j, and c_j is in B_i. Note that bc(G) is also a bigraph. Fig. C.1 illustrates the block-cutpoint tree for the given graph.

The block-cutpoint tree is used to order the sequence in which the partitions of a block are combined with the partitions of blocks previously partitioned. This sequence is dictated by the following rule: a block B_i is eligible for partitioning if at most one cutpoint of B_i is an element of the node set of some unpartitioned block. We base this rule on the following result.

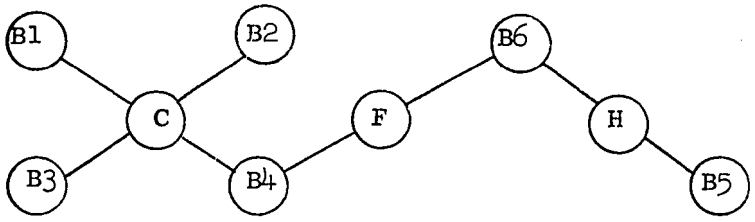
The nodes in a partitioned block B_i, with the possible exception of cutpoint c_j of B_i, are contained in no unpartitioned block. We refer to these nodes as "isolated nodes" since a cluster consisting entirely of these nodes is never modified in future steps of the partitioning process. As a result, the Isolated Set Theorem states that we can select the optimal-valued partition whose cluster containing node c_j is of weight w, where w=WEIGHT[c_j], WEIGHT[c_j]+1,...,W. Therefore a maximum of W of the partitions generated in the partitioning of B_i must be kept for future use in the partitioning process. These partitions represent the optimal-valued partitions of the subgraph whose nodes are in the blocks previously partitioned. We now describe the procedure used to generate and combine partitions:



(a) Graph with cutpoints C,F, and H



(b) Blocks of graph



(c) Block-cutpoint tree of graph

Figure C.1 -- The block-cutpoint tree

(1) Assign each node of the graph to be partitioned a unique identifier.

(2) Create the block-cutpoint graph bc(G).

(3) Choose some node of bc(G) that corresponds to a block as root. Form the directed tree bc'(G).

(4) Select a branch node c_k of bc'(G) all of whose sons are leaf nodes.

(a) Select some son of c_k. This node corresponds to a block of G, B_i, with cutpoint c_k in its node set.

(b) Label the nodes of B_i with integer labels $1,2,\ldots,n_i$ using the labeling algorithm developed in Chapter IV. Here n_i equals the number of nodes in B_i. Note that this labeling is independent of the labeling employed in partitioning another block of G.

(c) Partition B_i with the basic partitioning algorithm (Fig. 2.11) with one modification: If some node with label k is a cutpoint c_j, upon completion of step k find if previously-partitioned blocks also contained node c_j. If so, then there are a maximum of W optimal partitions of the subgraph consisting of the nodes in these blocks. Let this set of partitions be denoted by P_{c_j}. The partitions in P_k, the set of partitions generated on the kth step of the partitioning of block B_i, are then combined with the partitions in set P_{c_j}.

The combination of P_{c_j} and P_k results in a set denoted by $P_{k'}$. Since all nodes in clusters of the partitions of P_{c_j} are isolated nodes, except c_j, the maximum size of $\text{CONN}(k)_{max}$ remains unchanged and $|P_{k'}|_{max} = |P_k|_{max}$.

We now describe the process of combining the partitions in the sets P_k and P_{c_j}:

(i) Let an x-adjacency of node k be denoted by $p_{x,k}$ where $p_{x,k} \in P_k$. The partition in P_{c_j} with a cluster of weight y containing node c_j is denoted by p_{y,c_j}.

(ii) Combine $p_{x,k}$ and p_{y,c_j} by concatenating the clusters of each partition. The result is a partition $p_{z,k'}$ with clusters consisting of the unmodified clusters of $p_{x,k}$ and p_{y,c_j}, with the exception of the two clusters containing node c_j and node k. These two clusters are merged into one whose weight is given by

$$z=x+y-\text{WEIGHT}[c_j].$$

The set of nodes in the merged cluster includes the union of the set of nodes in both clusters containing node c_j (node k) with the identifier of node c_j replaced by its local label k.

(iii) $\text{VALUE}[p_{z,k'}]=\text{VALUE}[p_{x,k}]+\text{VALUE}[p_{y,c_j}]$

(d) When B_i is partitioned, delete the labels for each node of B_i and replace them with the identifier assigned in step (1).

(e) Delete B_i from bc'(G). If the number of nodes in B_i equals n_i, then $P_{c_k}=P_{n_i}$. If B_i was the last son of branch node c_k then delete c_k from bc'(G) also and select another branch node of bc'(G) whose sons are leaf nodes.

(f) Go to (a) until only the root B_r remains. Partition B_r using steps (a) through (d) above. Choose the optimal partition of those associated with B_r. This is the optimal partition of the graph.

B. GROWTH RATE

The growth in computation time is summarized in Table C.1. The number of operations is dominated by the operations required to partition a block of the graph (step 4c). The growth rate of the computation time to partition a graph G therefore varies asymptotically as

$$n(\bar{n}^{\,2})(p \log_2 p)$$

where p is the maximum number of partitions generated in partitioning a block of the graph G and $\bar{n}$ is the maximum number of nodes in a block of G.

The maximum storage requirements of the algorithm occur during steps 4c(i), 4c(ii), and 4c(iii) since Wp partitions may be generated during these steps. The growth in storage is therefore proportional to Wp.

C. EXAMPLE

We now give an example of the above procedure. The graph to be partitioned is shown in Fig. C.2(a).

STEP 1

Assign unique identifiers to nodes (Fig. C.2(b)).

STEP 2

Find bc(G) (Fig. C.1).

STEP 3

Form rooted tree bc'(G) by selecting node B_6 as root (Fig. C.3).

STEP 4

Select branch C since sons B_1, B_2, B_3 are all leaf nodes.

(a) Select B_1 for partitioning.

(b) Label B_1 (Fig. C.4(a)).

(c) Partition B_1 with algorithm outlined in Fig. 2.11:

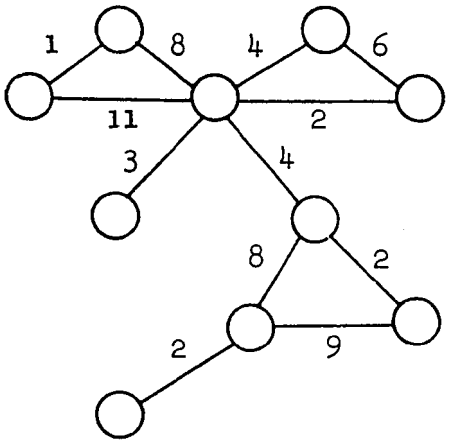
Table C.1 -- Number of operations to partition an n node graph with k cutpoints

STEP	OPERATIONS/STEP***	TOTAL OPERATIONS
1	$c_1 n$	$c_1 n$
2	$c_2 n^2$	$c_2 n^2$
3	$c_3 n$	$c_3 n$
4a	$c_4 n$	$c_4 kn$
4b	$c_5 \bar{n}^3$*	$c_5 k\bar{n}^3$
4c	$c_6 \bar{n}^2 (p \log_2 p)$**	$c_6 k\bar{n}^2 (p \log_2 p)$
4c(i,ii, and iii)	$c_7 \bar{n}$ Wp	$c_7 k\bar{n}$ Wp
4d	$c_8 \bar{n}$	$c_8 k\bar{n}$
4e	$c_9 n^2$	$c_9 kn^2$
4f	c_{10}	kc_{10}

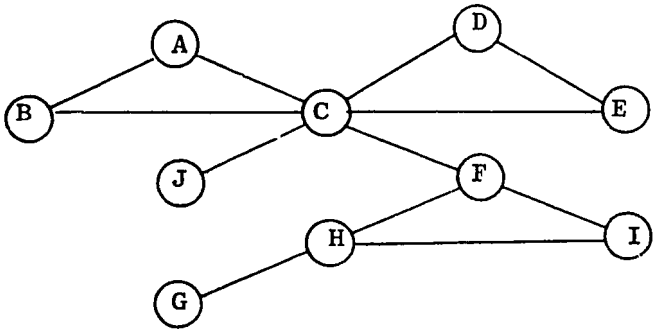
* $\bar{n}$=maximum number of nodes in a block of the graph

** p=maximum number of partitions generated in creating partitions of a block of the graph

*** The constants $c_1, c_2, \ldots, c_{10}$ are implementation dependent



(a) Graph to be partitioned



(b) Nodes of graph are assigned identifiers

Figure C.2 -- Example of partitioning process for graph with cutpoints

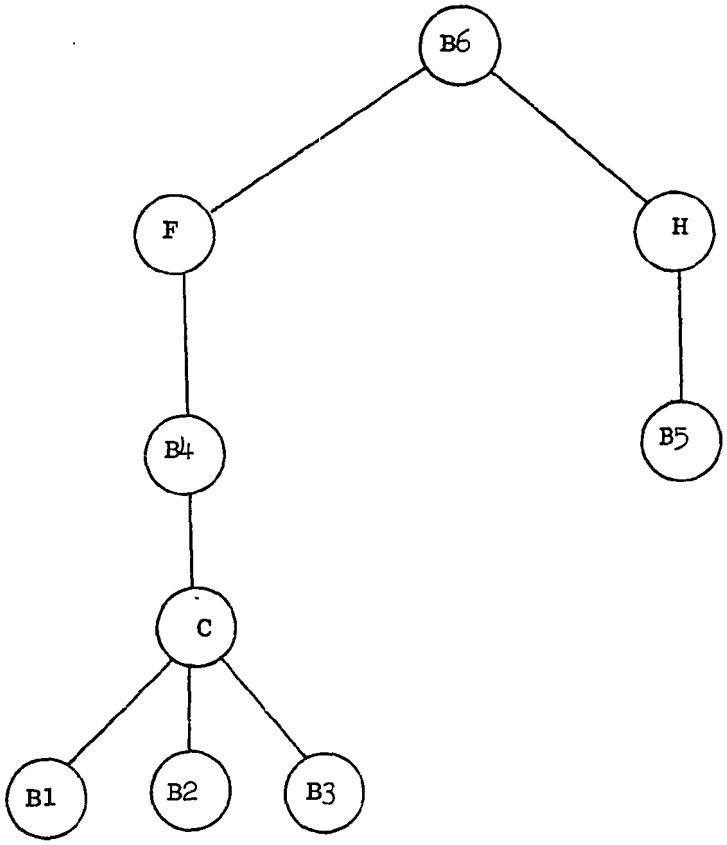
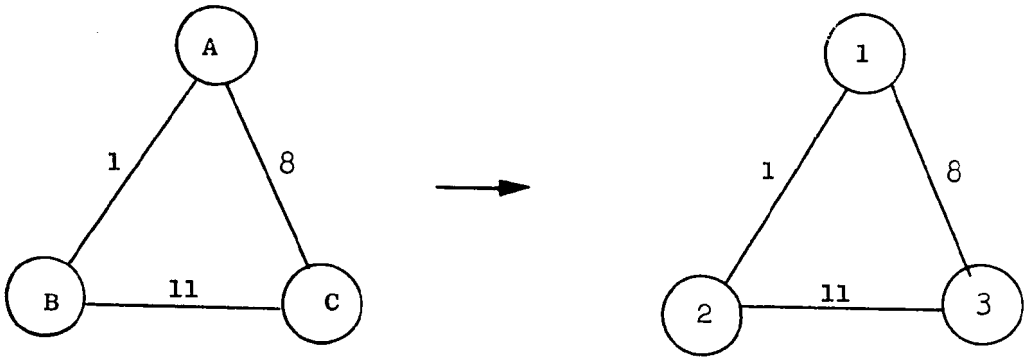
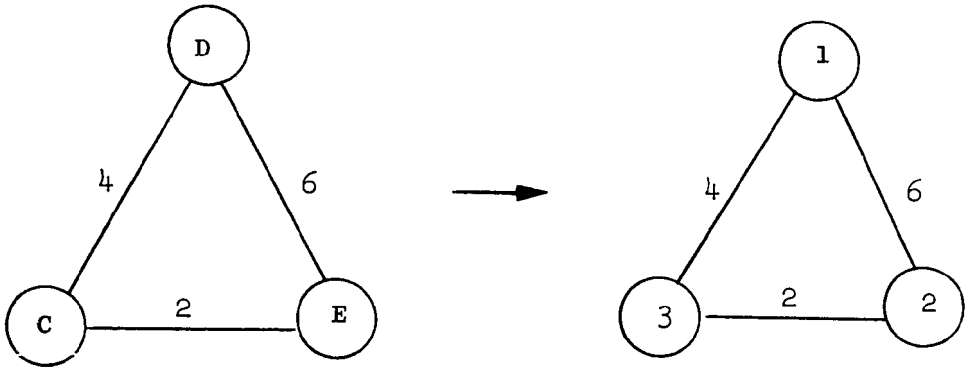


Figure C.3 -- Rooted tree bc'(G) derived from block-cutpoint graph bc(G)



(a) Labeling of B1



(b) Labeling of B2

Figure C.4 -- Examples of block labeling

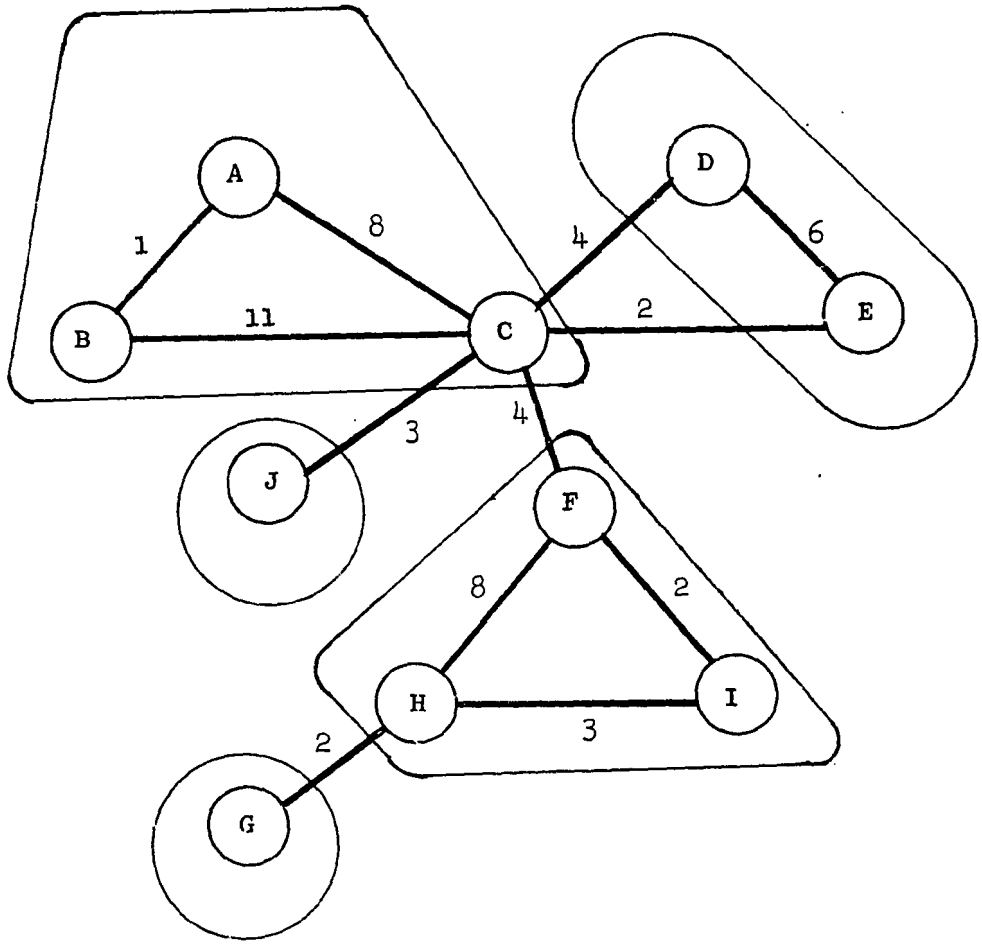


Figure C.5 -- Resulting partition of graph of Fig. C.2

Results --

$P_3=\{p_{1,3},p_{2,3},p_{3,3}\}$

$p_{1,3}=(1,2)(3)$ VALUE=1

$p_{2,3}=(2,3)(1)$ VALUE=11

$p_{3,3}=(1,2,3)$ VALUE=20.

(d) Delete labels and replace with node identifiers:

$P_c=\{p_{1,C},p_{2,C},p_{3,C}\}$

$p_{1,C}=(A,B)(C)$ VALUE=1

$p_{2,C}=(A)(C,B)$ VALUE=11

$p_{3,C}=(A,B,C)$ VALUE=20.

(e) Delete B_1 from tree.

Iteration 2:

(a) Select son B_2 of branch node C.

(b) Label B_2 (Fig. C.4(b)).

(c) Partition B_2:

$P_3=\{p_{1,3},p_{2,3},p_{3,3}\}$

$p_{1,3}=(1,2)(3)$ VALUE=6

$p_{2,3}=(2)(1,3)$ VALUE=4

$p_{3,3}=(1,2,3)$ VALUE=12.

Since node 3=node C, and node C is a cutpoint contained in a previously partitioned block B_1, combine P_3 with P_c:

$\underline{p_{1,3'}}$

combine $p_{1,3}$ and $p_{1,C}$: (A,B)(3)(1,2)

VALUE=7

$p_{2,3'}$

combine $p_{1,3}$ and $p_{2,C}$: (1,2)(3,B)(A)

VALUE=17

combine $p_{2,3}$ and $p_{1,C}$: (A,B)(1,3)(2)

VALUE=5

$p_{3,3'}$

combine $p_{1,3}$ and $p_{3,C}$: (A,B,3)(1,2)

VALUE=26

combine $p_{2,3}$ and $p_{2,C}$: (1,3,B)(2)(A)

VALUE=15

combine $p_{3,3}$ and $p_{1,C}$: (1,2,3)(A,B)

VALUE=13.

After deleting suboptimal partitions:

$P_{3'}=\{p_{1,3'},p_{2,3'},p_{3,3'}\}$

$p_{1,3'}$=(A,B)(3)(1,2) VALUE=7

$p_{2,3'}$=(1,2)(3,B)(A) VALUE=17

$p_{3,3'}$=(A,B,3)(1,2) VALUE=26

(d) Delete labels:

$P_C=\{p_{1,C},p_{2,C},p_{3,C}\}$

$p_{1,C}$=(A,B)(D,E)(C) VALUE=7

$p_{2,C}$=(A)(B,C)(D,E) VALUE=17

$p_{3,3}$=(A,B,C)(D,E) VALUE=26.

(e) Delete B_2 from bc'(G).

Iteration 3:

(a) B_3 is the remaining son of C.

(b,c,d) The generation of these partitions is similar to the above, hence:

$P_C=\{p_{1,C}, p_{2,C}, p_{3,C}\}$

$p_{1,C}=(A,B)(D,E)(C)(J)$ VALUE=7

$p_{2,C}=(B,C)(A)(D,E)(J)$ VALUE=17

$p_{3,C}=(A,B,C)(D,E)(J)$ VALUE=26

(e) Delete B_3 and C.

Iteration 4:

(a) Select node F since it has one son that is a leaf node, B_4.

(b,c,d) Resulting partitions:

$p_{1,F}=(A,B,C)(D,E)(J)(F)$ VALUE=26

$p_{2,F}=(A,B)(D,E)(C,F)(J)$ VALUE=11

$p_{3,F}=(B,C,F)(A)(D,E)(J)$ VALUE=21

(e) Delete nodes B_4 and F.

Iteration 5:

(a) Select node H since it has son B_5.

(b,c,d) Results:

$p_{1,H}=(G)(H)$ VALUE=0

$p_{2,H}=(G,H)$ VALUE=2

(e) Delete nodes B_5 and H.

Iteration 6:

The only node of bc'(G) left is the root, B_6. It is partitioned with the result that the optimal partition of G is

(A,B,C)(D,E)(J)(F,I,H)(G) VALUE=39.

Fig. C.5 shows this partition impressed upon the graph of Fig. C.2.

LIST OF REFERENCES

[1962] G. M. Adel'son-Vel'skii and E. M. Landis, "An algorithm for the organization of information," Dokl. Akad. Nauk SSSR, Mathemat., vol. 146, pp. 263-266, 1962.

[1965] C. C. Foster, "Information storage and retrieval using AVL trees," Proc. ACM Natl. Conf., pp. 292-305, 1965.

[1966] P. C. Gilmore and R. E. Gomory, "The theory and computation of knapsack functions," Oper. Res., vol. 14, pp. 1045-1074, Nov.-Dec. 1966.

[1967] J. Goldberg et al, Logic Design Techniques for Propagation Limited Networks, Stanford Research Institute, Rep. AFCRL-68-0002, pp. 5-51, Nov. 1967.

[1969] F. Harary, Graph Theory. Reading, Mass.: Addison-Wesley, 1969, pp. 36-37.

[1967] F. S. Hillier and G. J. Lieberman, Introduction to Operations Research. San Francisco, Cal.: Holden-Day, 1967, pp. 243-244.

[1971] H. Hopcroft and R. Tarjan, "Planarity testing in V log V steps," Computer Science Dept., Stanford University, Stanford, Cal., Rep. CS-71-201, Feb. 1971.

[1970] P. A. Jensen, "Optimal network partitioning," Opns. Res., vol. 19, pp. 916-932, Jul.-Aug. 1970.

[1969] B. W. Kernighan, "Some graph partitioning problems related to program segmentation," Ph.D. thesis, Princeton Univ., Princeton, N.J., Jan. 1969.

[1971] B. W. Kernighan, "Optimal Sequential Partitions of graphs," JACM, vol. 18, pp. 34-40, Feb. 1962.

[1962] E. L. Lawler, "Electrical assemblies with a minimum number of interconnections," IRE Trans. on Elect. Computers, pp. 86-88, Feb. 1962.

[1969] E. L. Lawler, K. N. Levitt and J. Turner, "Module clustering to minimize delay in digital networks," IEEE Trans. on Computers, vol. C-18, pp. 45-57, Jan. 1969.

[1968] C. L. Liu, Introduction to Combinatorial Mathematics. New York: McGraw-Hill, 1968, p. 187.

[1969] F. Luccio and M. Sami, "On the decomposition of networks in minimally interconnected subnetworks," IEEE Trans. on Circuit Theory, vol. CT-16, pp. 184-188, May 1969.

[1971] P. H. Oden, R. L. Russo and P. K. Wolff, "A heuristic procedure for the partitioning and mapping of computer logic graphs," IEEE Trans. on Computers, vol. C-20, pp. 1455-1462, Dec. 1971.

[1970] H. S. Stone, "An algorithm for module partitioning," JACM, vol. 18, pp. 182-195, Jan. 1970.

[1972] H. S. Stone, Introduction to Computer Organization and Data Structure. New York: McGraw-Hill, 1972, pp. 277-289.