J. A. Lukes

Efficient Algorithm for the Partitioning of Trees

Abstract: This paper describes an algorithm for partitioning a graph that is in the form of a tree. The algorithm has a growth in computation time and storage requirements that is directly proportional to the number of nodes in the tree. Several applications of the algorithm are briefly described. In particular it is shown that the tree partitioning problem frequently arises in the allocation of computer information to blocks of storage. Also, a heuristic method of partitioning a general graph based on this algorithm is suggested.

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 the different subgraphs is minimal. This type of graph partitioning problem arises in a variety of forms in computer systems, e.g., the clustering of logic circuits onto integrated circuit chips and the mapping of computer information onto physical blocks of storage.

No computationally efficient algorithm is known to exist for the partitioning of a general graph. However, partitioning algorithms have been described [1-8] that can be computationally efficient, given special features of the graph to be partitioned.

Here we restrict our attention to connected, acyclic graphs, or *trees*, and describe a dynamic programming algorithm for the partitioning of a tree. This partitioning algorithm has a growth in computation time and storage requirements directly proportional to the number of nodes in the graph. The ability to partition a tree with integer-weighted nodes and multivalued edges has not been considered in the literature. An algorithm [2] has been reported that partitions a special type of tree with a growth in computation of $n(\log_2 n)$ for an n-node tree; the edges of the tree must, however, assume a rather restricted set of values.

In this paper we first define the partitioning problem, as well as some terms and concepts useful in its characterization. Next we describe the tree partitioning algorithm and illustrate it by an example. We then consider possible applications of the algorithm. In particular we show that a tree partitioning problem of the type described here can arise in the allocation of data in hierarchical files to physical blocks of storage. Also, we suggest a possible heuristic procedure based on the algorithm for the partitioning of a general graph.

Definitions and basic concepts

Assume a tree $T = (V, E)$ with node set V and edge set E, as shown in Fig. 1. A *partition* of T is defined as a collection of k clusters of nodes $\{c_i\}$, $i = 1, 2, \cdots, k$, such that

$$\bigcup_{i=1}^{k} c_i = V;$$

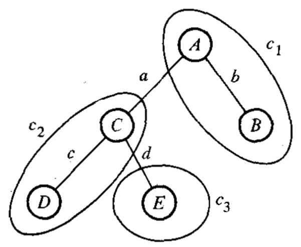
$$c_i \cap c_j = \varnothing \text{ for all } i \neq j.$$

A nonnegative integer *weight* w_i is associated with each node i of T. A weight constraint W is imposed on each cluster of T such that the sum of the weights of the nodes of any cluster does not exceed W.

An edge (i, j) of T is said to be *cut* by a partition of T if nodes i and j are in different clusters. A positive value v_{ij} is associated with each edge (i, j) of T. The *value* of a partition of T is equal to the sum of the values of the edges of T that are within its clusters (intracluster edges); the *cost* of a partition of T is equal to the sum of the values of the edges of T that are cut by the partition of T (intercluster edges). Thus the value plus the cost of a partition of T is equal to the sum of the values of the edges of T.

An *optimal partition* of T, $p_T\,(\text{opt}) = \{c_1, c_2, \cdots, c_k\}$, is one in which each cluster c_i satisfies the weight constraint

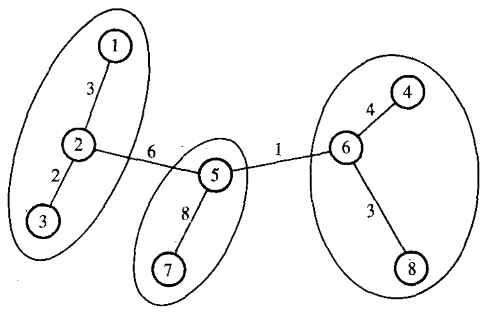
$$\sum_{j \in c_j} w_j \leq W,$$



$T = (V,E)$
$V = \{A,B,C,D,E\}$
$E = \{a,b,c,d\}$

Partition $= \{c_1, c_2, c_3\}$
$c_1 = \{A,B\}$
$c_2 = \{C,D\}$
$c_3 = \{E\}$

Figure 1 A partition of the tree $T = (V,E)$, where the weight constraint is 2 and all nodes are assumed to have unit weight.



Partition $p = (1,2,3)\ (4,6,8)\ (5,7)$
Value of $p = 3+2+4+3+8$
Cost of $p = 6+1$

Figure 2 Value and cost of a partition.

and in which cost

$\sum v_{ij}$ is minimal,

where $i \in c_f, j \in c_g, f, g = 1, 2, \cdots, k$, and $f \neq g$. An equivalent property of an optimal partition of G is one that satisfies the weight constraint and in which the value of T

$\sum_{i,j \in c_f} v_{ij}$ is maximal,

where $f = 1, 2, \cdots, k$.

To identify the different nodes of a tree, the tree is labeled by the assignment of a unique integer to each node, as in Fig. 2. To represent a partition we use a collection of lists in which 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 representation of a partition with this cluster is (1, 3, 5, 6) (2, 4,) (7).

Each cluster formed by a partition of tree T represents a subtree of T. However, we restrict our attention to partitions of T each of whose clusters forms a connected subtree of T, where *connected* means that every pair of nodes in the subtree is joined by a path [9]. We can apply this "connectivity" constraint [7] because the clusters of an optimal partition of any connected graph can be modified by forming from each cluster c_i a set of clusters each of which consists of a connected subgraph contained in c_i. Since the original partition was optimal and the new partition costs no more than the original, such a modification results in an optimal partition each of whose clusters is a connected subgraph.

For notational convenience, we change the given tree T into a *directed* and *ordered* tree T', as in Fig. 3. A directed, ordered tree is defined [10] to be a finite set T' of one or more nodes such that:

1. There is a distinct node called the *root* of T',
2. The remaining nodes (excluding the root) are separated into $m \geq 0$ disjoint sets $T'_1, T'_2, \cdots, T'_m$, and each of these sets is in turn a directed, ordered tree.

The trees $T'_1, T'_2, \cdots, T'_m$ are called the *subtrees* of the root, where T'_1 is the first subtree, T'_2 the second, etc.

The particular directed, ordered tree used does not affect the growth in computational complexity of the algorithm. By convention we form T' by selecting node 1 as the root of T' and the sons of node 1 become those nodes adjacent to (sharing an edge with) node 1. The sons of node 1 are ordered by increasing label value. If node i is a son of node 1, those nodes adjacent to node i (excluding node 1) are again ordered as the sons of node i by increasing label values. This process is repeated for each node in T resulting in the ordered, directed tree T'.

Partitioning algorithm

The basis of the tree partitioning algorithm is a dynamic programming technique that takes advantage of a basic property of a tree, its acyclic nature, to find a globally optimal partition based on local information. The algorithm generates the optimal partition of the tree T' by finding the partitions of increasingly larger subtrees of T' until the subtree that is partitioned is T' itself.

The first step in the partitioning algorithm is to generate the trivial partitions of the leaf nodes of T'. We then determine that set of nodes in T' all of whose subtrees have been partitioned. Assume that there is a node in this set with the label x. We generate the partitions of x by means of the following sequence of steps:

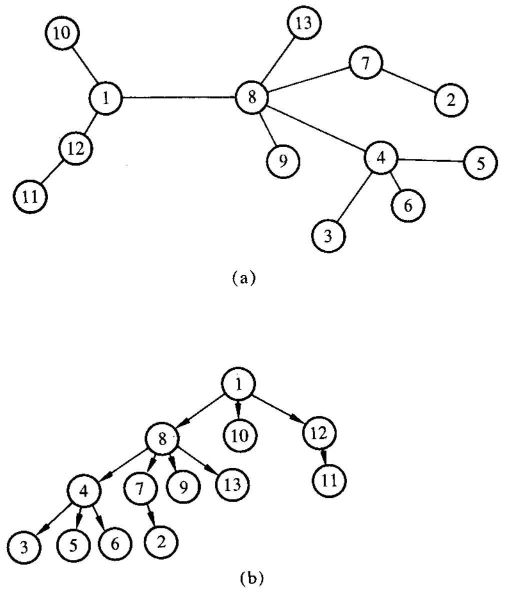


Figure 3 Transformation of tree into a directed tree. (a) Tree T; (b) directed, ordered tree T'.

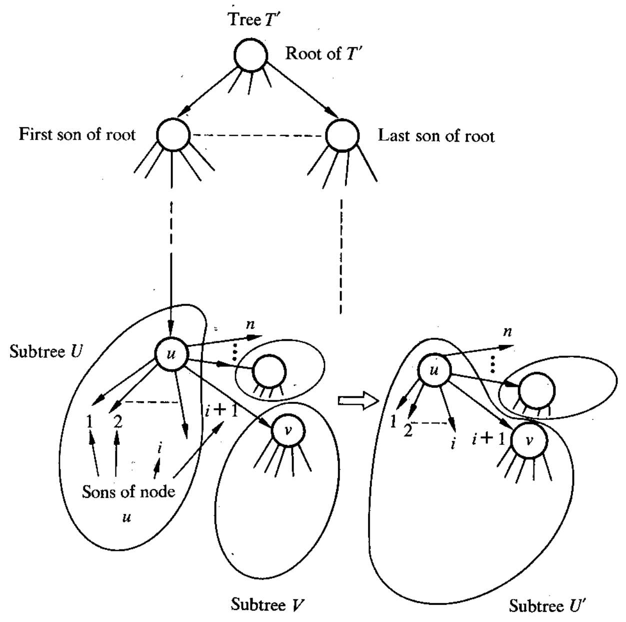


Figure 4 Generation of partitions by combining the partitions of two subtrees.

1. Find the partitions of node x and its first subtree.
2. Combine these partitions and the partitions of the second subtree of x to generate the partitions of the subtree composed of node x and its first two subtrees.
3. Combine the partitions created in step 2 with the partitions of the third subtree of x. The result is the set of partitions of the subtree composed of node x and its first three subtrees.
4. Continue this procedure such that on the $(i + 1)$st step the partitions of the subtree composed of node x and its first $i + 1$ subtrees are generated by combining the partitions generated on step i with the partitions of the $(i + 1)$st subtree of x.
5. Finally we reach a point in the algorithm when the tree with root x is partitioned.

As a result of the partitioning of the tree with root x, it may happen that the node that is the father of x becomes qualified as a node all of whose subtrees are partitioned. If such is the case, we add the father of node x to this set.

Upon finishing the partitioning of the tree with root node x, we remove node x from the set of nodes sharing the property that all of their subtrees are partitioned. Another node from this set is selected, and the tree for which this node is the root is partitioned in the manner described above. At some point in the algorithm, the set of nodes each of whose subtrees is partitioned is exhausted, whereupon we have generated the optimal partition of T'.

Before this algorithm can be considered to be practical, however, we must first find some orderly method of generating the partitions of a subtree consisting of some node u and its first $i + 1$ subtrees, given the partitions of the subtree composed of u and its first i subtrees and the partitions of the $(i + 1)$st subtree. Next, we observe that the number of partitions of a subtree can become very large. For example, consider the simple tree structure consisting of a node of T' all of whose sons are leaf nodes. The number of partitions of this tree such that the weight and connectivity constraints are both observed is

$$\sum_{i=0}^{W-1} \binom{k-1}{i}$$

where k = number of nodes in the tree and W = the weight constraint. For $w > k/2$, this bound grows as 2^k. We solve this problem by eliminating a large number of partitions of each subtree in the partitioning algorithm.

Consider some subtree of T', U in Fig. 4, which in general consists of a node u and its first i subtrees. In the proof of optimality to come later we will show that we need carry only a maximum of W partitions of U along in the partitioning process, where W is the weight constraint. If w is the weight of node u, this set of partitions is denoted by u_i $(i = w, w + 1, \cdots, W)$, where each partition u_i is distinguished from the other partitions of U by the properties:

1. Partition u_i is a partition of U whose cluster containing node u is of weight i,
2. Partition u_i is that partition of all those with property 1) whose value is maximal.

We now consider a method of generating the partitions of a subtree U', given the partitions of subtrees U and V, where the nodes of U and V comprise the nodes of U'.

In Fig. 4 we have two subtrees, U and V, whose partitions have been generated previously. To generate the partitions of U', the subtree created by combining U and V, we use either of two operations:

1. Concatenate the clusters of a partition of U, u_i, with those of a partition of V, v_j, merging the nodes in the clusters containing nodes u and v into a single cluster.
2. Concatenate the clusters of u_i and v_j.

We observe that any other combination of the partitions u_i and v_j violates the connectivity constraint.

Let us use the notation $C[u_i, v_j]$ to denote the partition created by the first operation above. The weight of the cluster of $C[u_i, v_j]$ containing node u is $i + j$, and its partition value is the sum of the value of u_i, v_j, and edge (i, j). The partition created by the second operation can also be denoted by the above notation if we define v_0 to be the partition of V that has the maximal value of all partitions of V. (If the second operation were used to combine u_i with any partition v_j, the resulting partition could never have a value greater than $C[u_i, v_0]$. Thus, we only consider the combination $C[u_i, v_0]$ in forming some u_i' using the second operation.) Then $C[u_i, v_0]$ represents a partition of U' whose cluster containing node u is of weight i and whose value is equal to the sum of the values of u_i and v_0.

We note that the assumption is made that all nodes in T (hence in T') are integer-weighted. Therefore, the partition u_k' of U' is selected from the set $\{C[u_1, v_{k-1}], C[u_2, v_{k-2}], \cdots, C[u_{k-1}, v_1], C[u_k, v_0]\}$, where we assume nodes u and v have unit weight for clarity of notation. (See Fig. 5.) The partition $C[u_a, v_b]$ in this set with maximal value is selected as u_k'.

We now summarize the steps associated with the partitioning algorithm.

Step 1 Label the tree T, and form the directed, order tree T'.

Step 2 For each leaf node u with weight w, form the partition $u_w = u_0 = (u)$. The value of this partition is zero. For all nodes v of T' that are branch nodes (nodes having one or more sons), initialize $(v) = v_j$ with value zero; here j is the weight of node v.

Step 3 Select some node x all of whose sons are leaf nodes, and form the optimal partitions for each weight equal to or less than the weight constraint of the subtree whose root is node x. To form these optimal partitions, follow these steps:

a. Let $i = 1$.
b. Form $x_j' = C[x_a, y_b]$ (for $j = w, w + 1, \cdots, W$), where the operator $C[x_a, y_b]$ forms partitions by either of the operations defined above; the particular partition $C[x_a, y_b]$ chosen is that of maximal value. Here y is the ith son of node x and $a + b = j$, where $w \leq a \leq W$ and $0 \leq b \leq W$. The weight of node x is w, and the weight constraint is W.
c. Make all $x_j = x_j'$. If $i =$ number of sons of node x, go to Step 4. Else, let $i = i + 1$ and go to Step 3 (b).

Step 4 Denote by x_0 the partition of the subtree whose root is x that has maximal value from the set $\{x_w, x_{w+1}, \cdots, x_W\}$.

Delete the sons of node x from the tree T'. If node x is the root of T', then x_0 represents the optimal partition of T' (hence of T). Otherwise, go to Step 3.

Proof of optimality

Consider some optimal partition q of the tree T'. Assume that each cluster of q contains a connected subtree of T'. As a consequence, we can separate the clusters of q into three groups:

1. Those clusters comprised of nodes in the subtrees whose roots are the first i sons of some node u,
2. A single cluster containing node u,
3. All other clusters.

Let us now proceed to delete from the clusters of q all those nodes (except node u) not in a subtree whose root is one of the first i sons of node u. The resulting clusters comprise a partition $\hat{u}$ of the subtree U of Fig. 4. We now claim that if the cluster of $\hat{u}$ containing node u is of weight j, then the value of $\hat{u}$ is equal to that of u_j.

Let us assume that the value of $\hat{u}$ is less than that of u_j. We can then generate a partition p whose value exceeds that of q by the amount that u_j exceeds $\hat{u}$. We do so as follows:

1. Make the clusters of p containing nodes not in subtree U equal to the clusters of q that are in the third group above.
2. Add the remaining nodes of T' (i.e., those nodes not in a cluster of $\hat{u}$ or in one of the group 3 clusters of q) to the cluster containing node u. We note that these nodes are precisely the nodes in the single cluster comprising group 2 of q not found in subtree U.

Clearly then q cannot be optimal if $\hat{u}$ does not have equal value to u_j.

In conclusion, the optimal partition of T' whose clusters form connected subgraphs can never be generated from some partition $\tilde{u}$ of subtree U, where $\tilde{u}$ has the following properties:

1. Its cluster containing node u is of weight j,
2. Its value is less than u_j.

We can then ignore all such partitions in generating the optimal partition of T'.

• *Growth rate*
Consider the number of steps required to form all possible partitions in combining the subtrees U and V of Fig. 4 to create the partitions of U'. There are a maximum of W partitions u_j' representing the set of optimal partitions of U'. Each such partition is chosen from a maximum of j partitions $C[u_a, v_b]$, where $a+b=j$ and $1 \leqq a$, $0 \leqq b$. The maximum number of partitions that must be considered in forming the u_j' is then given by the series

$$1+2+3\cdots+W=W(W+1)/2.$$

In forming the optimal partition of T', we are required to combine the partitions of some U with the partitions of some V once for every edge in the tree. Since an n-node tree has $n-1$ edges, the computational complexity grows as W^2n and is independent of the particular structure of the tree under consideration.

Example
We now illustrate the algorithm just described by means of an example. The tree of Fig. 6(a) is to be partitioned for a weight constraint of three. Each node is assumed to have unit weight, and the edges have the values shown.

Figure 6(b) shows the directed and ordered tree T' resulting from the labeling of T. We illustrate the steps in forming the optimal partition below, where the subscripts indicate either weight or an optimal partition and V indicates value:

Initialize:

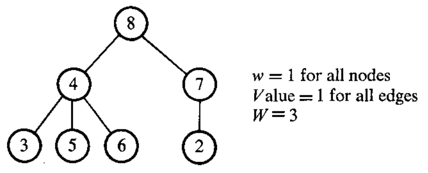
$1_1=(1)$	$V[1_1]=0$
$2_1=(2)$	$V[2_1]=0$
$3_1=3_0=(3)$	$V[3_0]=0$
$4_1=4_0=(4)$	$V[4_0]=0$
$5_1=5_0=(5)$	$V[5_0]=0$

To form partitions of the subtree whose root is 2, the first iteration is:

$2_1'=C[2_1, 3_0]=(2)(3)*$ with value $=0$.

*Optimal partition of the collection $C[u_a, v_b]=u_j$, where $a+b=j$.

$$2_2'=\begin{cases}C[2_1,3_1]=(2,3)* & \text{with value}=4\\ C[2_2,3_0] \text{ does not exist since } 2_2 \text{ cannot yet occur.}\end{cases}$$



Let $8_{12}=(8,4)(3)(5)(6)$	$V[8_{12}]=1$
$7_1=(2)(7)$	$V[7_1]=0$
$7_0=(2,7)$	$V[7_0]=1$

Then

$C[8_{12},7_1]=(7,8,4)(2)(3)(5)(6)$
$V[C[8_{12},7_1]]=2$

$C[8_{12},7_0]=(8,4)(3)(5)(6)(7,2)$
$V[C[8_{12},7_0]]=2$

Figure 5 Subtree combining operation.

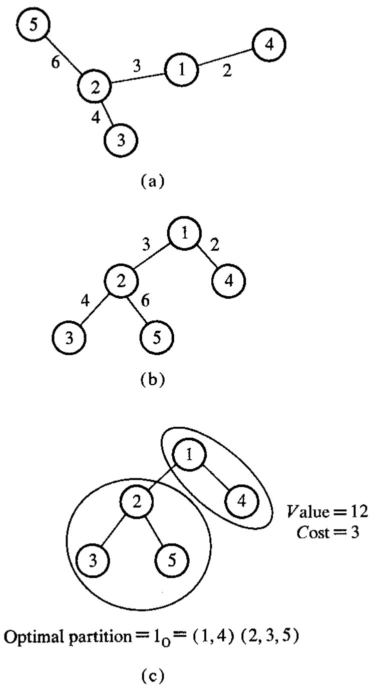


Figure 6 Optimal partitioning of a tree. (a) Tree T; (b) tree T'; (c) partition of T.

$2_3'=$ cannot occur since no previously generated partition of subtree 2 or 3 can form $2_3'$.

The second iteration is:

$2_1'=C[2_1, 5_0]=(2)(3)(5)*$ with value $=0$.

$$2_2'=\begin{cases}C[2_1,5_1]=(2,5)(3)* & \text{with value}=6\\ C[2_2,5_0]=(2,3)(5) & \text{with value}=4.\end{cases}$$

$$2_3'=\begin{cases}C[2_1,5_2] \text{ does not exist since } 5_2 \text{ cannot occur}\\ C[2_2,5_1]=(2,3,5)* \quad \text{with value}=10\\ C[2_3,5_0] \text{ does not exist since } 2_3 \text{ does not yet exist.}\end{cases}$$

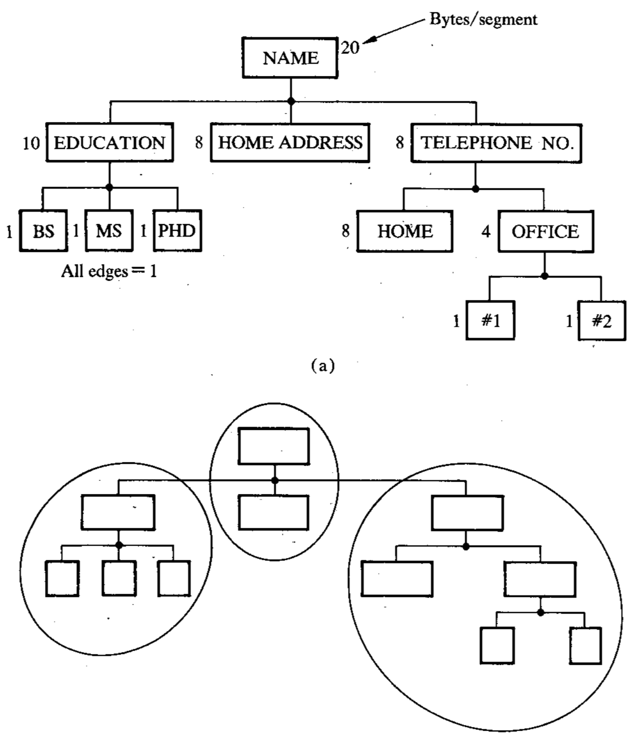


Figure 7 Application of tree partitioning. (a) Logical relationship of data in a hierarchical file; (b) allocation of data to 32-byte blocks.

Therefore

$$\begin{array}{ll} 2_0 = (2, 3, 5) & V[2_0] = 10 \\ 2_1 = (2)(3)(5) & V[2_1] = 0 \\ 2_2 = (2, 5)(3) & V[2_2] = 6 \\ 2_3 = 2_0 & V[2_3] = 10. \end{array}$$

To form the partitions of the subtree whose root is 1, the first iteration is

$$1_1' = C[1_1, 2_0] = (1)(2, 3, 5)^* \quad \text{with value} = 10.$$

$$1_2' = \begin{cases} C[1_1, 2_1] = (1, 2)(3, 5)^* & \text{with value} = 3 \\ C[1_2, 2_0] \text{ does not occur since } 1_2 \text{ has not yet occurred.} \end{cases}$$

$$1_3' = \begin{cases} C[1_1, 2_2] = (2, 5, 1)(3)^* & \text{with value} = 9 \\ C[1_2, 2_1] \text{ does not exist since } 1_2 \text{ does not yet exist} \\ C[1_3, 2_0] \text{ does not exist since } 1_3 \text{ does not yet exist.} \end{cases}$$

The second iteration is

$$1_1' = C[1_1, 4_0] = (1)(2, 3, 5)(4)^* \quad \text{with value} = 10$$

$$1_2' = \begin{cases} C[1_1, 4_1] = (1, 4)(2, 3, 5)^* & \text{with value} = 12 \\ C[1_2, 4_0] = (1, 2)(3, 5)(4) & \text{with value} = 3 \end{cases}$$

$$1_3' = \begin{cases} C[1_1, 4_2] \text{ does not exist since no } 4_2 \text{ exists.} \\ C[1_2, 4_1] = (1, 2, 4)(3, 5) & \text{with value} = 5 \\ C[1_3, 4_0] = (1, 2, 5)(3)(4)^* & \text{with value} = 9. \end{cases}$$

Therefore the optimal partition of the given tree is $1_0 = 1_2 = (1, 4)(2, 3, 5)$. Figure 6(c) illustrates this partition.

Tree partitioning problems

Graphs in the form of trees occur frequently in the representation of computer information. Certain linked lists, segments of nonrecursive programs, and hierarchical file structures are several examples of such tree structures. The tree partitioning problem arises when information must be allocated to blocks of memory whose capacity is limited. Kernighan [3] noted that the placement of program segments into pages in memory hierarchies is such a partitioning problem. Here the tree nodes are the segments, a node weight equals the number of bytes in a segment, and the edges represent a probable transition from one segment to another and are assigned values equal to the expected transitions between segments. The problem is then reduced to one of allocating nodes to clusters (pages) so as to reduce the expected number of page faults while executing the program.

A similar example of a tree partitioning problem is the distribution of the data of a hierarchical file structure into blocks of memory on a secondary storage device. One is given a logical data structure whose nodes consist of data segments and directory information and whose edges imply a logical relationship of data segments. If the expected transitions from node to node of a data structure are known, the structure assumes the form of a tree with weighted nodes and edges whose values equal the expected internode transitions. The objective then is to allocate nodes of the tree to finite capacity storage blocks. Since a search operation is required for each block of information in secondary storage, it is advantageous to cluster information referenced together frequently into the same blocks. Distributing the nodes in blocks such that the sum of the expected transitions between blocks is minimized results in the minimum number of interblock transitions for the hierarchical file structure. Figure 7 illustrates this problem.

We note that every connected graph has a number of embedded trees within it. For example, Fig. 8 shows a graph and one of its embedded trees. A heuristic algorithm for the partitioning of a general graph can be formulated based upon the tree partitioning algorithm, where the partitioned tree is an embedded tree of the given graph. First, an embedded tree T is determined from the given graph G. Those edges of G not in this tree are then considered to be cut. The embedded tree is partitioned, resulting in a partition of the graph. The cost of the resulting partition has a value z that lies in the range

$$y \leq z \leq x + y,$$

Where x = sum of edges of G not in T and y = sum of edges cut by the tree partition. This range results from the observation that one or more of the edges in G, but not T, may lie in a cluster of the resulting partition of T.

A connected graph may have many embedded trees, so that enumerating all embedded trees in order to find the optimal partition can become a formidable task. One technique (Fig. 9) that may be used to ease this problem is to find the maximal spanning tree of G, since this minimizes the sum of the edges removed from G. The resulting partition has a value such that the values of x and y are minimized, yet the sum $x + y$ may not be minimal because edges deleted in forming tree T may later be found in the cluster of a partition of T. No useful method of predicting those edges of G that may be included in a cluster of an optimal partition of the embedded tree T has been found. Figure 9 illustrates the selection of a suboptimal partition of a graph G using the above technique.

Summary

We have described a computationally efficient algorithm for the partitioning of a tree with integer-weighted nodes and edges with positive values. Several applications of the algorithm in the allocation of computer information to physical storage space are suggested. We also describe the use of this algorithm in finding a suboptimal partition of any connected graph.

A possible extension of this algorithm is to find a computationally efficient method of determining an optimal partition of any graph G based upon the partitioning of one of the embedded trees of G. Since the tree partitioning algorithm grows linearly with the number of tree nodes, an efficient method of selecting the embedded tree of G that results in the optimal partition is required.

References

1. P. A. Jensen, "Optimal Network Partitioning," *Oper. Res.* **19**, 916 (1970).
2. B. W. Kernighan, *Some Graph Partitioning Problems Related to Program Segmentations*, PhD. thesis, Princeton University, Princeton, N. J., January, 1969.
3. B. W. Kernighan, "Optimal Sequential Partitions of Graphs," *J. ACM* **18**, 34 (1971).
4. E. L. Lawler, "Electrical Assemblies With a Minimum Number of Interconnections," *IRE Trans. Elect. Computers* **C-11**, 86 (February 1962).
5. E. L. Lawler, K. N. Levitt, and J. Turner, "Module Clustering to Minimize Delay in Digital Networks," *IEEE Trans. Computers* **C-18**, 47 (1969).

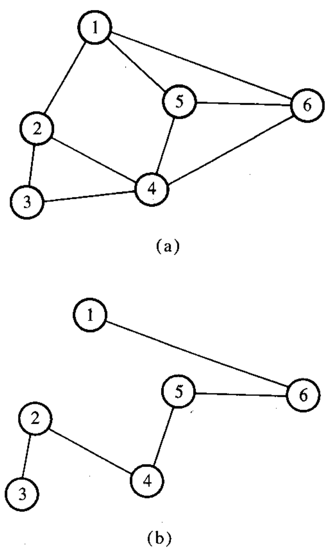
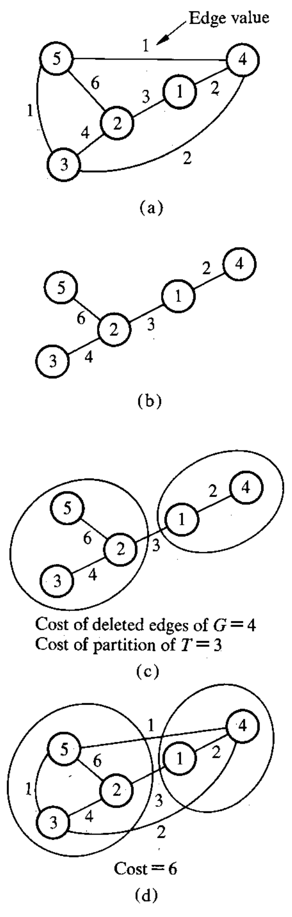


Figure 8 An embedded tree. (a) Graph G; (b) embedded tree of G.

Figure 9 Heuristic graph partitioning. (a) Graph G; (b) embedded tree T (maximal spanning tree); (c) resulting partition of T; (d) partition of G.



6. F. Luccio and M. Sami, "On the Decomposition of Networks in Minimally Interconnected Subnetworks," *IEEE Trans. Circuit Theory* **CT-16,** 184 (1969).
7. J. A. Lukes, "Combinatorial Solutions to Partitioning Problems," *Digital Systems Laboratory Report No. 32*, Stanford University, Stanford, California, June 1972.
8. H. S. Stone, "An Algorithm for Module Partitioning," *J. ACM* **18,** 182 (1970).
9. F. Harary, *Graph Theory*, Addison-Wesley Publishing Co., Inc., Reading, Mass., 1969.
10. D. E. Knuth, *The Art of Computer Programming* **1,** Addison-Wesley Publishing Co., Inc., Reading, Mass., 1968.

Received March 3, 1973; revised December 17, 1973

The author is located at the IBM Systems Development Division laboratory on Skyport Drive, San Jose, California 95114.