OPTIMAL CHAIN PARTITIONS OF TREES

Jayadev MISRA
Department of Computer Science, University of Texas at Austin,
Austin, Texas 78712, USA

and

R. Endre TARJAN
Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory,
University of California at Berkeley, Berkeley, CA 94720, USA

Received 5 May 1975, revised version received 13 June 1975

Algorithm, priority queue, chain, tree, dynamic programming

A tree $T = (V, E)$ is an undirected graph with vertex set $V$ and edge set $E$ such that $T$ is connected and contains no cycles. In a tree there is a unique simple path between any two vertices. A rooted tree $(T, r)$ is a tree $T$ with a distinguished vertex $r$ called the root. If $v$ and $w$ are vertices in a rooted tree $(T, r)$, we say $v$ is an ancestor of $w$ and $w$ is a descendant of $v$ (denoted by $v \rightarrow w$) if $v$ is contained in the path from $r$ to $w$. By convention $v \rightarrow v$ for all vertices $v$. If $v \rightarrow w$ and $\{v, w\}$ is an edge of $T$, we say $v$ is the father of $w$ and $w$ is a son of $v$ (denoted by $v \rightarrow w$).

It is useful to have a numbering of rooted tree vertices such that each vertex has a number smaller than its father. One such numbering, which is easy to compute, is a postorder numbering [4]. A postorder numbering of the vertices of a rooted tree $(T, r)$ is any numbering generated by the following algorithm:

```
procedure POSTORDER (T, r);
begin
    procedure SEARCH(w);
    begin
        for w such that v \rightarrow w do SEARCH(w);
        NUMBER(w) := i := i+1;
        end SEARCH;
    i := 0;
    SEARCH(r);
end POSTORDER;
```

A chain partition of a rooted tree $(T, r)$ is a collection $E'$ of edges such that, for any vertex $v$, $E'$ contains at most one edge $\{v, w\}$ with $v \rightarrow w$. Any chain partition $E'$ can be uniquely written as $E' = \bigcup_{i=1}^{k} P_i$, where $P_i$ is a set of edges which define a simple path in $T$ such that for any two vertices $v$ and $w$ on $P_i$, either $v \rightarrow w$ or $w \rightarrow v$, and where the edges of $P_i$ have no vertices in common with those of $P_j$ for $i \neq j$.

Given a rooted tree $(T, r)$, a non-negative cost $c_1(v)$ associated with each vertex, an (unrestricted) real-valued cost $c_2(v, w)$ associated with each edge, and a maximum cost $m \geq \max_{v \in V} c_1(v)$, we would like to find a chain partition $C = \bigcup_{i=1}^{k} P_i$ of maximum total edge cost satisfying $\sum_{v \in P_i} c_1(v) \leq m$ for all $i$. Such a chain partition we call an optimal chain partition.

* Research sponsored by National Science Foundation Grant GI-35604X1 and by a Miller Research Fellowship, at University of California, Berkeley, and by National Science Foundation Grant GI-36473X at Stanford University.
The optimal chain partition problem and similar problems occur when trying to divide a computer program optimally into pages and in other contexts where some kind of clustering is desired. See for instance [2, 3]. Suppose we represent a computer program as a directed graph, with each vertex \( v \) representing a block of code of size \( c_1(v) \) and each edge \((u, w)\) representing a transfer of control with associated cost \( c_2(u, w)\). We desire a partition of the program into blocks not exceeding size \( m \) such that the total cost of inter-block jumps is minimum. Thus the (avoided) total cost of jumps within blocks is maximum. This problem is NP-complete for arbitrary directed graphs [1], even if all \( c_1(v) \) and all \( c_2(u, w) \) are one. We give an efficient algorithm for trees.

If \( m \geq \sum_{u \in V} c_1(u) \) there is a very simple algorithm for trees. Let \( C(T) \) contain, for each vertex \( v \), a single edge \((v, w)\) of positive cost such that \( v \rightarrow w \) and \((v, w)\) has maximum cost among \((v, x)\) such that \( v \rightarrow x \). (If no edge \((v, w)\) with \( v \rightarrow w \) has positive cost, then \( C(T) \) contains no edge for vertex \( v \).) Obviously \( C(T) \) is an optimal chain partition of \((T, r)\) if \( m \geq \sum_{u \in V} c_1(u) \). Furthermore it is easy to construct \( C(T) \) in \( O(E) \) time if \( |V| = n \).

The general case for trees is somewhat harder. Henceforth assume that \( V = \{1, 2, \ldots, n\} \) where \( \text{NUMBER}(i) = i \) defines a postorder numbering of the vertices. (The above procedure computes a postorder numbering in \( O(n) \) time. Then we can identify vertices by their postorder numbers.)

Let \( v \) be any vertex of \( T \). The set of vertices \( \{w | v \rightarrow w\} \) defines a subtree \( T_v \) of \( T \). For all \( w \) such that \( v \rightarrow w \), let \( f_w(v) \) be the maximum total edge weight of a chain partition \( \bigcup_{i=1}^{k} P_i \) of \( T_v \). Such that the set of vertices in \( P_i \) is exactly \( \{x | v \rightarrow x \} \) and \( \sum_{u \in P_i} c_1(u) \leq m \) for \( 1 \leq i \leq k \). (The path \( P_i \) need not satisfy the vertex constraint.) Let \( g_w(v) = \sum_{u \rightarrow v \rightarrow w} c_1(x) \). Let \( f(v) = \max (f_w(v) + g_w(v) \leq m) \); this is finite since \( g_w(v) \leq m \).

Several facts are obvious from these definitions. The cost of an optimal chain partition of \((T, r)\) is \( f(r) = f(n) \). Also \( f_w(v) = \sum_{v \rightarrow x} f(x) \) and \( g_w(v) = c_1(v) \). Last, \( u \rightarrow v \) and \( v \rightarrow w \) imply \( g_w(u) = g_w(v) + c_1(u) \) (and thus \( g_w(u) \leq g_w(v) \)), and \( f_w(u) = f_w(v) + c_2(u, v) + c_2(v, u) + \sum_{x \rightarrow u, x \neq v} f(x) \).

These “dynamic programming” equations are enough to allow efficient calculation of \( f(n) \). For \( i \) running from \( 1 \) to \( n \) (i.e. working from sons to fathers) we calculate \( f(i) \), and \( f_j(i) \) and \( g_j(i) \) for all vertices \( j \) in a subset of \( T_i \) large enough to support succeeding calculations. To implement these calculations we associate a set \( Q(i) \) (the “queue” of \( i \)) with every vertex \( i \). Each element \( x \) of \( Q(i) \) will have three associated parameters:

\[ I(x): \text{a vertex } j \text{ in } T_i; \]
\[ F(x): \text{the value of } f_j(i); \]
\[ G(x): \text{the value of } g_j(i). \]

To compute the desired values efficiently we need the following operations.

(i) \( \text{INSERT}(i, x) \): inserts the element \( x \) into \( Q(i) \).

Time required: \( O(1) \).

(ii) \( \text{UNION}(i, j) \): moves all the elements in \( Q(j) \) into \( Q(i) \), leaving \( Q(j) \) empty.

Time required: \( O(\log |Q(i)| + \log |Q(j)| + 1) \).

(iii) \( \text{MAXF}(i) \): returns an element in \( Q(i) \) with maximum \( F \)-value, and deletes the element from \( Q(i) \).

Time required: \( O(1) \).

(iv) \( \text{ADDF}(i, z) \): adds the value \( z \) to the \( F \)-variable of all the elements in \( Q(i) \).

Time required: \( O(1) \).

(v) \( \text{ADDG}(i, z) \): adds the value \( z \) to the \( G \)-variable of all the elements in \( Q(i) \).

Time required: \( O(1) \).

These operations can be implemented to run in the given time bounds by using leftist trees, as shown in [5, 6]; operations (iv) and (v) put special kinds of nodes into the data structure.

In the algorithm given in fig. 1 all the queues are initially empty. The idea behind the queues is that after a vertex \( i \) is processed (by the outermost loop) but before its father is processed, \( Q(i) \) must contain an element for each vertex \( j \) in \( T_i \) such that \( g_j(i) \leq m \). (It may also contain some \( j \)'s such that \( g_j(i) > m \).)

The program stores, for each vertex \( i \), the value of \( f(i) \) and a value \( h(i) \) which denotes the last vertex on the path containing \( i \) in the optimal chain partition which is computed.

It is easy to see that OCP correctly computes \( f(i) \) for each \( i \) and hence correctly computes \( f(n) \). By using the values \( h(i) \), \( 1 \leq i \leq n \), it is easy to construct a directed chain partition having total edge weight \( f(n) \). The time required by OCP is dominated by the time spent in queue operations. There are \( 2n \) INSERT
algorithm OCP;
begin
for $i := 1$ until $n$ do
begin
  $s := \sum_{j \in \mathcal{F}(i)} f(j)$;
  let $x$ be a new queue element with parameters
  $I(x) = i$, $F(x) = s$, $G(x) = 0$;
  INSERT($i$, $x$);
  for $j$ such that $i \rightarrow j$ do
  begin
    ADDF($j$, $s-f(j) + c_2(i, j)$);
    QUNION($i$, $j$);
  end;
  ADDG($i$, $c_1(i)$);
  $x := \text{MAXF}(i)$;
  while $G(x) > m$ do $x := \text{MAXF}(i)$;
  $f(i) := F(x)$;
  $h(i) := I(x)$;
  INSERT($i$, $x$);
end
end OCP;

Fig. 1.

operations, $n-1$ QUNION operations, $n-1$ ADDF
operations, $n$ ADDG operations, and at most $2n$
MAXF operations (only $2n$ elements are added to
queues and hence only $2n$ can be deleted), so OCP
requires $O(n \log n)$ time total. The space requirements
are $O(n)$ (see [5, 6]).

Our thanks to the referee for his unusually perceptive
and helpful remarks.

References

simplified NP-complete problems, Sixth ACM Symp. on
[2] B.W. Kernighan, Optimal sequential partitions of graphs,
[3] J. Kral, To the problem of segmentation of a program,
Information Processing Machines, Research Inst. for
Mathematical Machines, Prague (1965) 140–149.
Fundamental Algorithms (Addison-Wesley, Reading,
Sorting and Searching (Addison-Wesley, Reading, Mass.,
1973), 150–152.
ERL-M591, Electronics Research Lab., Univ. of Cali-
fornia (1975).