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 $v \overset{*}{\to} w$) if v is contained in the path from r to w. By convention $v \overset{*}{\to} v$ for all vertices v. If $v \overset{*}{\to} w$ 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 $v \to w$).

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

‡ Research sponsored by National Science Foundation Grant GJ-35604X1 and by a Miller Research Fellowship, at University of California, Berkeley, and by National Science Foundation Grant GJ-36473X at Stanford University.

```
procedure POSTORDER (T, r);
  begin
    procedure SEARCH(v);
      begin
        for w such that v → w  do  SEARCH(w);
        NUMBER(v) := 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 $v \to w$. Any chain partition E' can be uniquely written as $E' = \cup_{i=1}^{k} P_i$, 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 $v \overset{*}{\to} w$ or $w \overset{*}{\to} v$, 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 $m \geqslant \max_{v \in V} c_1(v)$, we would like to find a chain partition $C = \cup_{i=1}^{k} P_i$ of maximum total edge cost satisfying $\Sigma_{v \text{ on } P_i} c_1(v) \leqslant m$ for all i. Such a chain partition we call an *optimal chain partition.*

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 (v, w) representing a transfer of control with associated cost $c_2(v, w)$. 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 $c_2(v, w)$ are one. We give an efficient algorithm for trees.

If $m \geqslant \Sigma_{v \in V} c_1(v)$ 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 $v \to w$ and $\{v, w\}$ has maximum cost among $\{v, x\}$ such that $v \to x$. (If no edge $\{v, w\}$ with $v \to w$ has positive cost, then $C(T)$ contains no edge for vertex v.) Obviously $C(T)$ is an optimal chain partition of (T, r) if $m \geqslant \Sigma_{v \in V} c_1(v)$. Furthermore it is easy to construct $C(T)$ in $O(E) = O(n)$ time if $|V| = n$.

The general case for trees is somewhat harder. Henceforth assume that $V = \{1, 2, ..., n\}$ where 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 $\{w | v \xrightarrow{*} w\}$ defines a subtree T_v of T. For all w such that $v \xrightarrow{*} w$, let $f_w(v)$ be the maximum total edge weight of a chain partition $\cup_{i=1}^{k} P_i$ of T_v such that the set of vertices in P_1 is exactly $\{x | v \xrightarrow{*} x \text{ and } x \xrightarrow{*} w\}$ and $\Sigma_{v \in P_i} c_1(v) \leqslant m$ for $2 \leqslant i \leqslant k$. (The path P_1 need not satisfy the vertex constraint.) Let $g_w(v) = \Sigma_{v \xrightarrow{*} x \xrightarrow{*} w} c_1(x)$. Let $f(v) = \max \{f_w(v) | v \xrightarrow{*} w$ and $g_w(v) \leqslant m\}$; this is finite since $g_v(v) \leqslant m$.

Several facts are obvious from these definitions. The cost of an optimal chain partition of (T, r) is $f(r) = f(n)$. Also $f_v(v) = \Sigma_{v \to x} f(x)$ and $g_v(v) = c_1(v)$. Last, $u \to v$ and $v \xrightarrow{*} w$ imply $g_w(u) = g_w(v) + c_1(u)$ (and thus $g_w(u) \geqslant g_w(v)$), and $f_w(u) = f_w(v) + c_2(u, v) + \Sigma_{u \to x, x \neq v} f(x)$.

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:

$I(x)$: a vertex j in T_i;
$F(x)$: the value of $f_j(i)$;
$G(x)$: the value of $g_j(i)$.

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

(i) INSERT (i, x): inserts the element x into $Q(i)$. Time required: $O(1)$.
(ii) QUNION(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) MAXF(i): returns an element in $Q(i)$ with maximum F-value, and deletes the element from $Q(i)$. Time required: $O(\log|Q(i)| + 1)$.
(iv) ADDF(i, z): adds the value z to the F-variable of all the elements in $Q(i)$. Time required: $O(1)$.
(v) 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 $g_j(i) \leqslant m$. (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 $h(i)$, $1 \leqslant i \leqslant n$, 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 := Σ_{i→j} 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 → j do
          begin
            ADDF(j, s−f(j) + c2(i, j));
            QUNION(i, j);
          end ;
        ADDG(i, c1(i));
        x := MAXF(i);
        while G(x) > m do x := 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

[1] M.R. Garey, D.S. Johnson and L.J. Stockmeyer, Some simplified NP-complete problems, Sixth ACM Symp. on Theory of Computing (1974) 47–63.

[2] B.W. Kernighan, Optimal sequential partitions of graphs, J. ACM 18 (1971) 34–40.

[3] J. Kral, To the problem of segmentation of a program, Information Processing Machines, Research Inst. for Mathematical Machines, Prague (1965) 140–149.

[4] D. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms (Addison-Wesley, Reading, Mass., 1968), 315–346.

[5] D. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Reading, Mass., 1973), 150–152.

[6] R. Tarjan, "Finding minimum spanning trees", Memo No. ERL-M501, Electronics Research Lab., Univ. of California (1975).