Bottleneck Multicast Trees in Linear Time.

Leonidas Georgiadis

***Abstract*— On a directed graph with arc costs and a given source node s, we consider the problem of computing multicast (Steiner) trees spanning any given node subset V, so that the maximum of the tree arc costs is minimized. We show that this problem can be solved by simply solving the bottleneck path problem, i.e., the problem of determining for each node $t \neq s$ a path from s to t so that the maximum of path arc costs is minimized. For the latter problem we provide an implementation of Dijkstra's algorithm that runs in linear time under mild assumptions on arc costs.**

I. Introduction

Consider a directed graph $G(N, A)$ with $n = |N|$ nodes and $m = |A|$ arcs, and let b_a be a real number cost associated with arc a. Given a source node s and a subset V of nodes, $s \notin V$, the min-max or bottleneck Steiner tree problem is to determine a directed tree routed at node s that spans all nodes in V (i.e., there is a directed path on the tree from s to any node $t \in V$), such that the maximum of the tree arc costs is minimized. This problem received early attention and it is still important as part of the design of multicast trees in communication networks either wired, with b_a representing the negative of link bandwidth [1], or wireless, with b_a representing node consumed power to reach a neighbor [2].

In fact, in a communication network setup, it may also be desirable to be able to *precompute* the multicast tree for any V, [1]. If such a precomputation can be made, it will be possible to quickly provide the required trees in a dynamic environment where V changes frequently. Of course, the number of sets V is $2^{n-1} - 1$ and in general precomputing all the trees will be prohibitive both in terms of running time and in terms of required storage space. However, it will be seen that for the problem at hand the situation is greatly simplified since all required trees can be obtained as subtrees of a single spanning tree.

For undirected graphs, [3] presents an algorithm that solves the bottleneck Steiner tree problem and runs in $O(m)$ time. However, the algorithm is not suitable for precomputation since it provides a Steiner tree for the specified set V only. The problem of finding a bottleneck spanning tree in directed graphs, i.e., the case $V = N - \{s\}$, has been addressed in [4]. The proposed algorithm can be extended to the case where V is any subset of $N - \{s\}$ and runs in $O(m \log n)$ time. This algorithm cannot be used for precomputing the optimal trees for any V. In [5] two algorithms are proposed for solving the bottleneck spanning tree problem, both of them extendable to any node subset V. The first is based on a modification of Dijkstra's algorithm and runs in $O(n \log n + m)$ time. The second runs in $O(m \log^* n)$ time, where $\log^* n$ is the iterated logarithm of n. However, the second algorithm cannot be used for precomputation.

In this paper we address the problem of precomputing bottleneck Steiner trees. We propose an implementation of Dijkstra's algorithm for solving the bottleneck path problem, which runs in $O(T(m))$ time where $T(m)$ is the time needed to sort the costs b_a, $a \in A$. Under mild assumptions on the costs, $T(m) = O(m)$. We then show that the optimal path tree obtained by this process, contains as subtrees a bottleneck Steiner tree for any node subset V and therefore it can be used for precomputation.

L. Georgiadis is with Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, GREECE. E-mail: leonid@eng.auth.gr .

II. Bottleneck Paths and Trees

A. Bottleneck Paths

Let a directed graph $G(N, A)$ be given and let $N(i)$ be the set of nodes adjacent to node i, i.e., $N(i) = \{j \in N : (i,\ j) \in N\}$. With arc $a = (i, j)$, we associate a real number cost, b_a. Given a source node s, we are interested in finding a simple path p_t^* from s to any node $t \in V$ such that the maximum of the path arc costs is minimized. Path p_t^* is called "bottleneck" path. More precisely, if $\mathcal{P}_t$ is the sets of simple paths from s to t, we want to solve the optimization problem.

Problem I (Bottleneck Paths to All Nodes):

$$\min_{p \in \mathcal{P}_t} \max_{a \in p} \{b_a\} \text{ for all } t. \tag{1}$$

Without loss of generality assume that $b_a \geq 0$, $a \in A$. This problem can be solved by a simple modification of Dijkstra's algorithm, which in generic form, [6], is shown in Figure 1(a). The algorithm maintains for each node $i \in N$ a distance label $d[i]$, which is initialized to infinity (a number larger that the maximum arc cost) except for node s, for which it is set $d(s) = 0$; in addition it maintains a set S whose labels are optimal (i.e., $d[i] = \min_{p \in \mathcal{P}_i} \max_{a \in p} \{b_a\}$) and smaller than or equal to the labels of the nodes in $\overline{S} = N - S$. The algorithm maintains also an array $pred[i]$ denoting upon completion the node j such that (j, i) is the last arc in the bottleneck path from s to i (with the convention, $pred[s] = null$). Array $pred[i]$ specifies a tree containing bottleneck paths from s to any node $t \in N - \{s\}$. A Fibonachi heap implementation of $\overline{S}$ results in $O(n \ln n + m)$ running time.

For Problem I, we present below an implementation of Dijkstra's algorithm whose running time is $O(T(m))$, where $T(m)$ is the time needed to sort the weights b_a, $a \in A$.

The implementation takes advantage of two simple observations. Namely

1) Let r_a be the ranking order of b_a, i.e., b_a is the r_ath smallest element in the set $\{b_a : a \in A\}$. Then, any

```
1.  algorithm Dijkstra
2.  begin
3.      S ; S N;
4.      d i  for each node i N;
5.      d s 0; pred s null;
6.      while |S| n do
7.          let i S be such that d i min d j : j S ;
8.          S S i ;
9.          S S i ;
10.          for each j N i
11.              if d j max d i , b i,j then do
12.                  d j max d i , b i,j and pred j i;
13.              end;
14.          end;
15.      end;
16.  end;
```

(a)

```
1.  algorithm Dijkstra Implementation
2.  begin
3.      obtain r a , a N from b a ;
4.      for each node i N
5.          F i .label , F i .address null;
6.      end;
7.      F s .label 0; F s .address insert L 0 , s ;
8.      pred s null; C 1;
9.      for l 0 to m and while C n do
10.         if get(L l ) null do
11.             i get(L l );
12.             F i .address null; C C 1;
13.             for each element g in K i
14.                 j g.node; a g.arc;
15.                 if F j .label max l, r a then do
16.                     if F j .value then do
17.                         delete(L F j .label , F j .address);
18.                     end;
19.                     F j .label max l, r a ;
20.                     F j .address insert(L F j .label , j);
21.                     pred j i;
22.                 end;
23.             end;
24.         end;
25.     end;
26. end;
```

(b)

Fig. 1. (a) Dijkstra Algorithm (b)Proposed Implementation

solution to

$$\min_{p\in\mathcal{P}_t} \max_{a\in p} \{r_a\} \text{ for all } t,$$

is a solution to (1). Indeed, if p an q are simple paths from s to t, then

$$\max_{a\in p} \{r_a\} \leq \max_{a\in q} \{r_a\} \text{ iff } \max_{a\in p} \{b_a\} \leq \max_{a\in p} \{b_a\}.$$

Hence we can replace b_a with r_a in Dijkstra's algorithm.

2) The non-infinite values taken by the labels $d[j]$ belong to the set $\{b_a : a \in A\}$.

Because of the first observation, we can replace b_a with r_a in Dijkstra's algorithm, while as will be seen, because of the second, we can implement line 7 of Dijksta's algorithm in Figure 1(a) by a simple one-time linear search of an $(m+1)$-element array containing the label values. As a result, an $O(T(m))$ time algorithm is obtained.

The proposed implementation maintains the array $pred[i]$ of the generic Dijkstra algorithm. In addition, we need the following data structures. We assume that arcs are indexed from 1 to m.

- An input array of arc costs, $b[a]$, $a = 1, ..., m$.
- An output array $r[a]$, $a \in A$ containing the rank of arc a.
- An array of node adjacency singly-linked lists, $K[i], i \in N$. An element g of list $K[i]$ contains two fields, $g.node$ and $g.arc$. Field $g.node$ denotes a node j in $N(i)$ and field $g.arc$ denotes the arc number corresponding to arc (i, j). Hence $r[g.arc]$ denotes the rank of the arc $(i, g.node)$.
- An array $L[l]$ of doubly-linked lists $0 \leq l \leq m$. During each iteration, list $L[l]$ contains the nodes that have nonpermanent label value l. Each array supports the following unit cost operations.
  - insert($L[l], i$): Inserts an element with value i in the list and returns the address of its location.
  - delete($L[l], \&i$): Deletes the element at location $\&i$.
  - get($L[l]$): Returns and deletes the first element in the list.
- An array $F[i]$, $i \in N - \{s\}$. Element $F[i]$ of the array contains two fields, $F[i].label$, $F[i].address$. Initially $F[i].label = \infty$ and $F[i].address$ is null. During the course of the algorithm, $F[i].label$ contains the (permanent or nonpermanent) label of node i and $F[i].address$ contains the address of the element in $L[F[i].label]$ that contains node i, provided that $F[i].label \neq \infty$ and the label of node i has not been permanently set. If $F[i].label = \infty$ at the end of the algorithm, then there is no directed path from s to node i, while when the label of node i is permanently set, $F[i].address$ is set to null.

The proposed algorithm is shown in Figure 1(b). The algorithm examines each of the $m + 1$ lists $L[l]$. $l = 0, .., m$. The nonempty list examined at each iteration of the loop in line 9 contains the nodes with the least nonpermanent labels and hence line 11 of the algorithm implements line 7 of the generic algorithm in Figure 1(a). Lines 13 to 23 implement lines 10 to 14 of the generic algorithm. If $F[j].label > \max\{l, r[a]\}$ (line 15), then the label of node j must be reduced. We therefore remove in line 17 node j from list $L[F[j].label]$ (provided of course that it has already been placed in the list, which happens only if $F[j].value < \infty$). Next, we update the label of node j in line 19. In line 20 we insert node j in the list $L[F[j].label]$ and save the address of the inserted element in $F[j].address$.

Regarding the complexity of the algorithm, the execution of line 3 takes $T(m)$ time. Observe next that whenever a "get" operation is performed in line 11, the label of a node is permanently set. Therefore, this operations are performed at most n times. The "for" loop in line 9 is executed at most $m+1$ times, and the code in the for loop in line 13 is executed at most m times. Since each of the operations within the latter "for" loop takes $O(1)$ time, the overall running time of the algorithms is $O(n + m + T(m)) = O(T(m))$.

B. Bottleneck Steiner Trees

We now address the bottleneck Steiner tree problem. Let $\mathcal{T}_V$ be the set of directed trees routed at s and spanning all nodes in $V \subseteq N - \{s\}$.

Problem II (Bottleneck Steiner Trees to All Subsets $V \subseteq N - \{s\}$):

$$\min_{T \in \mathcal{T}_V} \max_{a \in T} \{b_a\} \text{ for all subsets } V \subseteq N - \{s\}.$$

As part of the solution to Problem I, a tree T^* routed at s and spanning $N - \{s\}$ is obtained, with the property that the path p_t^* on T^* joining s to any node t, is a bottleneck path. For any set $V \subseteq N - \{s\}$, let T_V^* be the subtree of T^* consisting of the arcs of the bottleneck paths p_t^*, $t \in V$. The next lemma shows that T_V^* is a bottleneck tree for the set V.

Lemma 1: It holds

$$\max_{a \in T_V^*} \{b_a\} = \min_{T \in \mathcal{T}_V} \max_{a \in T} \{b_a\}.$$

Proof: Since $T_V^* \in \mathcal{T}_V$, it holds

$$\max_{a \in T_V^*} \{b_a\} \geq \min_{T \in \mathcal{T}_V} \max_{a \in T} \{b_a\}. \quad (2)$$

Assuming that

$$\max_{a \in T_V^*} \{b_a\} > \min_{T \in \mathcal{T}_V} \max_{a \in T} \{b_a\}, \quad (3)$$

we will arrive a a contradiction, which in lieu of (2) implies the lemma.

Under assumption (3), there is a tree $T \in \mathcal{T}_V$, such that

$$\max_{a \in T_V^*} \{b_a\} > \max_{a \in T} \{b_a\}. \quad (4)$$

Let $e \in T_V^*$ be an arc such that $b_e = \max_{a \in T_V^*} \{b_a\}$. There is a node $t \in V$ for which the path p_t^* from s to t in T_V^* passes through e. There is also a path q from s to t in T . For these paths we have because of (4),

$$\max_{a \in p_t^*} \{b_a\} = \max_{a \in T_V^*} \{b_a\} > \max_{a \in T} \{b_a\} \geq \max_{a \in q} \{b_a\}. \quad (5)$$

However, (5) contradicts the fact p_t^* is a bottleneck path for node t. ■

According to Lemma 1, problem II is solved by simply solving Problem I and obtaining the tree consisting of bottleneck paths to nodes in V. Hence the complexity of the solution to Problem II is again $O(T(m))$.

Figure 2 shows an example of the construction of a bottleneck-path tree. The numbers next to the arcs are the arc costs. All paths on the tree using arcs $\{(s,1),(1,2),(2,3),(1,4)\}$ and having source s, are bottleneck paths from s to each of the nodes $1,2,3,4$. If we need a bottleneck Steiner tree with source s and destination node set $V = \{2,4\}$, then we can simply take the subtree of the bottleneck-path tree consisting of the arcs $\{(s,1),(1,2),(1,4)\}$. Note that another bottleneck Steiner tree for the same set V is the path $\{(s,2),(2,3),(3,4)\}$. This path has arcs that do not belong to the bottleneck-path tree.

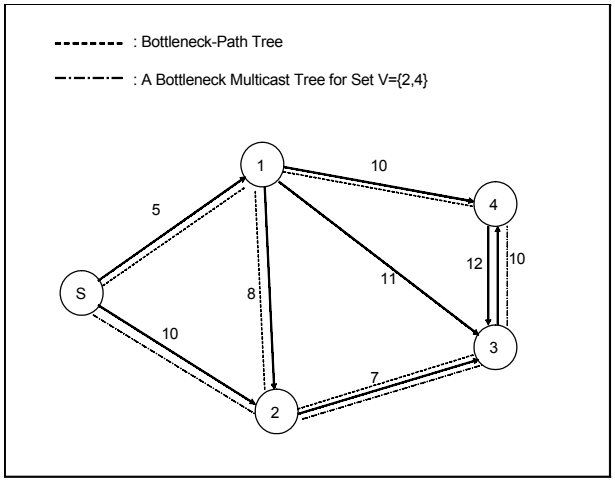

Fig. 2. Example of Bottleneck-Path Tree

III. Arc Costs and Algorithm Complexity

The complexity of the proposed implementation is in effect determined by the complexity of sorting the numbers b_a, $a \in A$. In fact, if b_a are integers and $|b_a| = O(m)$ for $a \in A$, then we can work directly with the array $b[a]$ instead of $r[a]$, and therefore we can eliminate altogether the sorting cost in line 3 of the algorithm in Figure 1(b). In this case we obtain an $O(m)$ worst-case running time algorithm using very simple data structures. If for some constant d, $|b_a| = O(m^d)$ for $a \in A$, then again $T(m) = O(m)$ using radix sort [7]. This case covers many applications.

In general, any complexity result obtained for the well-studied sorting problem, translates directly to a corresponding result for Problem II. For example, it was found in [8] that for unit-cost RAM with world length of w bits, if b_a take values in the range $0, ..., 2^w - 1$ where $w \geq \log m$, sorting can be achieved in $O(m \log\log m)$ time. Moreover, if $w \geq (\log m)^{2+\varepsilon}$ for some $\varepsilon > 0$, then sorting can be accomplished in linear expected time with a randomized algorithm. Under the same conditions, the same claims can be made for problem Problem II.

References

[1] R. Guerin, A. Orda, "Computing Shortest Paths for Any Number of Hops," *IEEE/ACM Transactions on Networking,* vol 10, no 5, October 2002, pp 613-620.

[2] I. Papadimitriou, L. Georgiadis, "Energy-aware Broadcasting in Wireless Networks," *WiOpt'03: Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, March 3-5, 2003, INRIA Sophia-Antipolis, France.

[3] C. W. Duin, A. Volgenant, "The partial sum criterion for Steiner trees in graphs and shortest paths," *European Journal of Operational Research,* 97 (1997) 172-182.

[4] P.M. Camerini, "The min-max spanning tree problem and some extensions," *Information Processing Letters*, Vol 7, Number 1, January 1978, pp 10-14.

[5] H. N. Gabow, R. E. Tarjan, "Algorithms for Two Bottleneck Optimization Problems," *Journal of Algorithms,* 9, 411-417 (1988).

[6] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms,* Mc-Grow Hill, MIT Press, 1990.

[8] A. Anderson, T. Hagerup, S. Nilsson, R. Raman, "Sorting in Linear Time?," *27th Annual ACM Symposium on the Theory of Computing* in Las Vegas, Nevada, May 1995.