JOURNAL OF ALGORITHMS 9, 411–417 (1988)

Algorithms for Two Bottleneck Optimization Problems

HAROLD N. GABOW*

Department of Computer Science, University of Colorado, Boulder, Colorado 80309

AND

ROBERT E. TARJAN†

Department of Computer Science, Princeton University, Princeton, New Jersey 08544 and AT&T Bell Laboratories, Murray Hill, New Jersey 07974

Received July 13, 1987; accepted November 19, 1987

A *bottleneck optimization problem* on a graph with edge costs is the problem of finding a subgraph of a certain kind that minimizes the maximum edge cost in the subgraph. The bottleneck objective contrasts with the more common objective of minimizing the sum of edge costs. We propose fast algorithms for two bottleneck optimization problems. For the problem of finding a bottleneck spanning tree in a directed graph of n vertices and m edges, we propose an $O(\min\{n \log n + m, m \log^* n\})$-time algorithm. For the bottleneck maximum cardinality matching problem, we propose an $O((n \log n)^{1/2} m)$-time algorithm. © 1988 Academic Press, Inc.

1. INTRODUCTION

Consider network optimization problems in which we are given a graph with edge costs and asked to find a subgraph of a certain kind that minimizes some function of the costs of the edges in the subgraph. The usual objective function is the sum of the edge costs. Another natural

*Research partially supported by the National Science Foundation Grant MCS-8302648.
†Research partially supported by the National Science Foundation Grant DCR-8605962.

objective function is the maximum of the edge costs. We shall call optimization problems with the former objective function *sum problems* and problems with the latter, *bottleneck problems*.

Some bottleneck problems can be solved asymptotically more efficiently than the corresponding sum problems. For example, Camerini [3] has observed that a bottleneck spanning tree in an undirected graph with n vertices and m edges can be computed in $O(m)$ time. The best known time bound to find a spanning tree minimizing the sum of edge costs is $O(m \log \beta(m, n))$, where $\beta(m, n) = \min\{i | \log^{(i)} n \leq m/n\}$ [9]. Here $\log^*$ is the iterated logarithm, defined by $\log^{(0)} x = x$, $\log^{(i+1)} x = \log \log^{(i)} x$, $\log^* x = \min\{i | \log^{(i)} x \leq 1\}$.

In this note we present two results of this kind, for two problems posed by the second author in a problems column [14]. For the problem of finding a bottleneck spanning tree in a directed graph, we give an $O(\min\{n \log n + m, m \log^* n\})$-time algorithm. The best known bound for finding a spanning tree minimizing the sum of edge costs is $O(n \log n + m)$ [9]. For the problem of finding a bottleneck maximum cardinality matching in an undirected graph, we give an $O((n \log n)^{1/2} m)$-time algorithm. This improves the $O(n^{1/2} m \log n)$-time algorithm of Bhat [1], which he stated only for bipartite graphs. The best known bound for finding a minimum total cost maximum cardinality matching depends on whether the problem graph is bipartite or not. In the former case (the *assignment problem*), such a matching can be computed in $O(n^2 \log n + nm)$ time [6], or in $O(n^{1/2} m \log(nC))$ time if all edge costs are integers of absolute value at most C [10]. In the latter case, such a matching can be computed in $O(n^2 \log n + nm \log \log \log_{2m/n} n)$ time [8], or in $O((n\alpha(m, n) \log n)^{1/2} m \log(nC))$ time if all edge costs are integers of absolute value at most C, where α is a functional inverse of Ackerman's function [11]. Whether there are even better bounds for the two bottleneck problems we consider remains an open question.

2. Bottleneck Spanning Trees in Directed Graphs

Let $G = (V, E)$ be a directed graph with n vertices and m edges. For ease in stating time bounds we assume that $m \geq n \geq 2$. Let s be a distinguished *root vertex* of G and for each edge (v, w) let $c(v, w)$ be a real-valued *cost*. We consider the problem of finding a spanning tree rooted at s (containing paths from s to all other vertices) whose maximum edge cost is minimum. We call such a tree a *bottleneck spanning tree*.

Camerini [3] has proposed an $O(m \log n)$-time algorithm for finding a bottleneck spanning tree. We shall describe an $O(\min\{n \log n + m, m \log^* n\})$-time method.

To obtain an $O(n \log n + m)$ bound, we note that Dijkstra's single-source shortest path algorithm [4], if modified slightly, will compute a bottleneck spanning tree. The algorithm grows a tree from the root s. It computes a parent $p(v)$ in the tree for each $v \neq s$. The algorithm maintains a collection of vertices F that are candidates for inclusion in the tree. Each vertex $v \in F$ has an associated cost $c(v)$ for inclusion, which is the minimum cost of an edge from a vertex already in the tree to v. Initially, $F = \{s\}$ and $c(s) = -\infty$. The algorithm consists of repeating the following step n times:

General Step. Select a vertex $v \in F$ with $c(v)$ minimum and delete it from F. For every edge (v, w), if w is neither in the tree nor in F, add w to F and define $c(w) = c(v, w)$ and $p(w) = v$; otherwise, if w is in F and $c(v, w) < c(w)$, replace $c(w)$ by $c(v, w)$ and define $p(w) = v$.

If a Fibonacci heap [6] is used to implement the frontier set F, this algorithm runs in $O(n \log n + m)$ time.

For sparse graphs we can obtain a better bound of $O(m \log^* n)$. For any real number λ, let $G(\lambda) = (V, \{(v, w) \in E | c(v, w) \leq \lambda\})$, and let $\lambda^* = \min\{\lambda | \forall v \in V$ there is a path in $G(\lambda)$ from s to $v\}$. To find a bottleneck spanning tree it suffices to compute λ^*, since any spanning tree in $G(\lambda^*)$ is a bottleneck spanning tree for G, and a spanning tree in $G(\lambda^*)$ can be found in $O(m)$ time.

We find λ^* by using repeated splitting to narrow the interval of possible values of λ. The number of intervals into which the current interval is split is a function $k(i)$ of the number of splits i that have taken place; as i increases so does $k(i)$. The algorithm maintains values λ_1 and λ_2 such that $\lambda_1 \leq \lambda^* \leq \lambda_2$. Initially λ_1 is the minimum edge cost, λ_2 is the maximum edge cost, and i, a count of the number of iterations, is zero. The algorithm consists of the following steps:

1. Replace i by $i + 1$. Let $S_0 = \{(v, w) \in E | c(v, w) \leq \lambda_1\}$ and $E_1 = \{(v, w) \in E | \lambda_1 < c(v, w) \leq \lambda_2\}$.

2. Partition E_1 into $k(i)$ subsets $S_1, S_2, \ldots, S_{k(i)}$, each of size $\lfloor |E_1|/k(i) \rfloor$ or $\lceil |E_1|/k(i) \rceil$, such that if $(v, w) \in S_i$ and $(x, y) \in S_{i+1}$, then $c(v, w) \leq c(x, y)$.

3. Find the minimum j such that $G_j = (V, S_0 \cup S_1 \cup S_2 \cup \cdots \cup S_j)$ is such that all vertices are reachable from s.

4. If $j = 0$, let $\lambda^* = \lambda_1$ and stop. Otherwise, replace λ_1 and λ_2, respectively, by the minimum cost and the maximum cost of an edge in S_j, and go to Step 1.

The correctness of this algorithm is obvious. Steps 1 and 4 each take $O(m)$ time per iteration. Step 2 can be done by repeated median-finding:

split E_1 into a lower half and an upper half, then split each half into halves and so on. (We shall choose $k(i)$ to be a power of two.) Since median-finding takes linear time [2, 17], step 2 takes $O(|E_1|\log k(i))$ time per iteration.

Step 3 takes $O(m)$ time using an incremental search. The search begins at s and advances only along edges in S_0. If the search stops before all vertices are reached, edges in S_1 become eligible for searching. In general, each time the search stops without having reached all vertices, edges in the next set S_j become eligible for searching. Implementation of such a search requires the maintainance, for each vertex v, of an adjacency list $A(v)$. Each vertex is in one of three states: *unlabeled*, *labeled*, *or scanned*. Initially s is labeled, all other vertices are unlabeled, $j = 0$, and $A(v) = \{w | (v, w) \in S_0\}$ for each vertex v. The search consists of the following steps:

3.1. If some vertex is labeled, select a labeled vertex v, mark it scanned, and go to step 3.2. Otherwise, if no vertex is unlabeled, stop. Otherwise go to step 3.3.

3.2. For every vertex $w \in A(v)$, if w is unlabeled, mark it labeled. Go to step 3.1.

3.3. Replace j by $j + 1$. For each edge $(v, w) \in S_j$, if v is unlabeled, add w to $A(v)$; otherwise, if w is unlabeled, mark w labeled. Go to step 3.1.

It remains to choose $k(i)$ and to analyze the total running time of the algorithm. We define $k(1) = 2$, $k(i) = 2^{k(i-1)}$ for $i \geq 1$. This choice guarantees that, in the ith iteration, $|E_1| = O(m/k(i-1))$, which implies that step 2 in the ith iteration takes $O((m/k(i-1))\log k(i)) = O(m)$ time. Thus the total time per iteration is $O(m)$. The number of iterations is $O(\log^* n)$, giving an overall time bound of $O(m \log^* n)$.

Almost the same algorithm will solve the following problem: given two vertices s and t in a directed graph with edge costs, find a path from s to t that minimizes the maximum cost of an edge on the path (or, equivalently, maximizes the minimum cost). We call such a path a *bottleneck shortest path*. This problem arises in the implementation of Edmonds and Karp's "maximum capacity augmentation" version of the Ford–Fulkerson maximum network flow algorithm [5]. The only change necessary in the algorithm above is to stop the incremental search as soon as t is labeled. Thus the bottleneck shortest path problem can be solved in $O(\min\{n \log n + m, m \log^* n\})$ time. If the graph is undirected, a variant of Camerini's algorithm [3] will compute a bottleneck shortest path in $O(m)$ time, as observed by Lesley Matheson (private communication, 1987).

If the edges are given in sorted order, a bottleneck shortest path tree can be found in $O(m)$ time. We merely run Step 3 of the above algorithm once, with $j = m$ and each S_i containing a single edge.

3. Bottleneck Maximum Cardinality Matchings

Let $G = (V, E)$ be an undirected graph. G is *bipartite* if V can be partitioned into two sets A and B such that every edge has one vertex in A and one vertex in B. A *matching* is a subset of edges, no two sharing a common vertex. A *maximum-cardinality matching* is a matching containing as many edges as possible. Suppose G has a real-valued *cost* $c(v, w)$ on each edge (v, w). We consider the problem of finding a maximum-cardinality matching whose maximum edge cost is minimum. We call such a matching a *bottleneck* matching. In the discussion to follow, we assume some familiarity with standard augmenting path methods for finding a maximum cardinality matching. (See e.g. [18].)

The bottleneck matching problem for the special case of bipartite graphs (the *bottleneck assignment problem*) has been considered by Gross [13], Garfinkel [12], and Bhat [1]. Garfinkel observes that a bottleneck assignment can be computed by using an incremental search method to find augmenting paths. Although Garfinkel gives no time bound, the resulting algorithm can be seen to run in $O(nm)$ time.

Bhat observes that a maximum cardinality matching algorithm in combination with binary search yields an $O(n^{1/2}m \log n)$-time algorithm for computing a bottleneck matching. We describe the method below; then we modify it to reduce the running time to $O(n \log n)^{1/2}m)$.

Algorithms for computing a maximum cardinality matching in $O(n^{1/2}m)$ time have been developed for the bipartite case by Hopcroft and Karp [15] and for the general case by Micali and Vazirani [16]. These algorithms have the important but often overlooked property that they can be used as approximation algorithms in the following sense: if M^* is a maximum cardinality matching, then in $O(km)$ time for any k either algorithm will compute a matching M such that $|M^*| - |M| \leq n/k$. This property is the key to our fast algorithm.

The first step in computing a bottleneck matching is to sort the edges by cost. This takes $O(m \log n)$ time. Let $e_1, e_2, \ldots, e_m$ be the edges, in nondecreasing order by cost. Using a maximum cardinality matching algorithm, we compute l, the size of a maximum cardinality matching. Then we use binary search to find the minimum value i^* of i such that the graph $G_i = (V, E_i = \{e_1, e_2, \ldots, e_i\})$ contains a matching of size l. To test whether a guess i for i^* is high or low, we use a maximum cardinality matching subroutine. The total running time of this method is $O(n^{1/2}m \log n)$.

We do better by using binary search to find a bottleneck matching that is only approximately of maximum cardinality. For a parameter k (to be chosen below), we compute a value i' such that the graph $G_{i'}$ has a matching size at least $l - n/k$, and G'_{i-1} does not have a matching of size l. To test whether a guess i for i' is high or low, we use the approximation version of the Hopcroft–Karp algorithm (on a bipartite graph) or the Micali–Vazirani algorithm (on a general graph). We compute a matching M in G_i such that $|M|$ is within n/k of maximum. If $|M| < l - n/k$, then the guess i is too low. The time to find i' using this approach is $O(km \log n)$. This gives us not only a value for i', but also a matching M of size at least $l - n/k$, all of whose edges have cost at most $c(e_{i'}) \leq c(e_{i^*})$.

The matching M can be augmented to form a bottleneck matching by finding a bottleneck minimum augmenting path, augmenting M accordingly, and repeating this at most n/k times. Each augmentation requires a search for an augmenting path. Such a search can be done in $O(m)$ time by using a standard method for finding an augmenting path (as described for example in [18] for general graphs) and making it incremental as in Section 2: we initialize i to be equal to i', let the search advance only along edges in E_i, and increase i by one each time the search terminates without finding an augmenting path. The total time to augment M to form a bottleneck matching is $O(nm/k)$.

The overall time to find a bottleneck matching is $O(mk \log n + nm/k)$. Choosing $k = (n/\log n)^{1/2}$ gives a time bound of $O(n \log n)^{1/2}m)$.

The same technique gives a bound of $O(n \log n)^{1/2}m)$ for finding a cardinality-k matching whose maximum edge cost is minimum, where k is an input parameter. The bottleneck matching problem arises in approximate weighted matching and in efficient implementation of Christofides' heuristic for the traveling salesman problem [7, 11]. Our approach to bottleneck matching can also be used in efficient algorithms for computing the density and arboricity of a graph [19].

References

1. K. V. S. Bhat, An $O(n^{2.5}\log_2 n)$ time algorithm for the bottleneck assignment problem, unpublished report, AT&T Bell Laboratories, Napierville, IL, 1984.
2. M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, Theoretical improvements in algorithmic efficiency for network flow problems, *J. Comput. System Sci.* **7** (1973), 448–461.
3. P. M. Camerini, The min–max spanning tree problem and some extensions, *Inform. Process. Lett.* **7** (1978), 10–14.
4. E. W. Dijkstra, A note on two problems in connexion with graphs, *Numer. Math.* **1** (1959), 269–271.
5. J. Edmonds and R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *J. Assoc. Comput. Mach.* **19** (1972), 248–264.

6. M. L. FREDMAN AND R. E. TARJAN, Fibonacci heaps and their uses in network optimization, *J. Assoc. Comput. Mach.* **34** (1987), 596–615.
7. H. N. GABOW, A scaling algorithm for weighted matching on general graphs, *in* Proceedings, 26th Annual IEEE Symp. on Found. of Comput. Sci., 1985, pp. 90–100.
8. H. N. GABOW, Z. GALIL, AND T. SPENCER, Efficient implementation of graph algorithms using contraction, *in* Proceedings, 25th Annual IEEE Symp. on Found. of Comput. Sci., 1984, pp. 347–357.
9. H. N. GABOW, Z. GALIL, T. SPENCER, and R. E. Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, *Combinatorica* **6** (1986), 109–122.
10. H. N. GABOW AND R. E. TARJAN, Faster scaling algorithms for network problems, *SIAM J. Comput.*, submitted; also Technical Report CS-TR-111-87, Department of Computer Science, Princeton University, Princeton, NJ, 1987.
11. H. N. GABOW AND R. E. TARJAN, unpublished manuscript, 1987.
12. R. S. GARFINKEL, An improved algorithm for the bottleneck assignment problem, *Oper. Res.* **19** (1971), 1747–1750.
13. O. GROSS, "The Bottleneck Assignment Problem," P-1630, The Rand Corporation, Santa Monica, CA, 1959.
14. L. J. GUIBAS (Ed.), Problems, *J. Algorithms* **6** (1985), 283–290.
15. J. E. HOPCROFT AND R. M. KARP, An $n^{5/2}$ algorithm for maximum matching in bipartite graphs, *SIAM J. Comput.* **2** (1973), 225–231.
16. S. MICALI AND V. V. VAZIRANI, An $O(\sqrt{|V|} \cdot |E|$ algorithm for finding maximum matching in general graphs, *in* Proceedings, 21st Annual IEEE Symp. on Found. of Comput. Sci., 1980, pp. 17–27.
17. A. SCHÖNHAGE, M. PATERSON, AND N. PIPPENGER, Finding the median, *J. Comput. System Sci.* **13** (1976), 184–199.
18. R. E. TARJAN, "Data Structures and Network Algorithms," Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
19. H. H. WESTERMANN, "Efficient Algorithms for Matroid Sums," Ph.D. dissertation, Department of Computer Science, University of Colorado, Boulder, CO, 1987.