COMBINATORICA 6 (2) (1986) 109—122

EFFICIENT ALGORITHMS FOR FINDING MINIMUM SPANNING TREES IN UNDIRECTED AND DIRECTED GRAPHS

H. N. GABOW*, Z. GALIL**, T. SPENCER*** and R. E. TARJAN

Received 23 January 1985
Revised 1 December 1985

Recently, Fredman and Tarjan invented a new, especially efficient form of heap (priority queue). Their data structure, the *Fibonacci heap* (or F-heap) supports arbitrary deletion in $O(\log n)$ amortized time and other heap operations in $O(1)$ amortized time. In this paper we use F-heaps to obtain fast algorithms for finding minimum spanning trees in undirected and directed graphs. For an undirected graph containing n vertices and m edges, our minimum spanning tree algorithm runs in $O(m \log \beta(m, n))$ time, improved from $O(m\beta(m, n))$ time, where $\beta(m, n) = \min \{i | \log^{(i)} n \leq m/n\}$. Our minimum spanning tree algorithm for directed graphs runs in $O(n \log n + m)$ time, improved from $O(n \log n + m \log \log \log_{(m/n+2)} n)$. Both algorithms can be extended to allow a degree constraint at one vertex.

1. Introduction

A *heap* (sometimes called a *priority queue*) is an abstract data structure consisting of a collection of *items*, each with a real-valued *key*, on which at least the following operations are possible:

make heap: Return a new, empty heap.

insert (x, h): Insert item x, with predefined key, into heap h, assuming that it is not already in h.

find min (h): Return an item of minimum key in heap h, without changing h.

delete min (h): Delete from heap h an item of minimum key and return it. If h is empty, return a special null item.

A variety of other operations on heaps are sometimes useful. These include the following:

meld (h_1, h_2): Return the heap formed by taking the union of item-disjoint heaps h_1 and h_2. This operation destroys h_1 and h_2.

* Research supported in part by National Science Foundation Grant MCS—8302648.
** Research supported in part by National Science Foundation Grant MCS—8303139.
*** Research supported in part by National Science Foundation Grant MCS—8300984 and a United States Army Research Office Program Fellowship, DAAG29—83—GO020.
AMS subject classification (1980):68 B 15, 68 C 05

decrease key (Δ, x, h): Decrease the key of item x in heap h by subtracting the non-negative real number Δ. This operation assumes that the location of x in h is known.

delete (x, h): Delete item x from heap h, assuming that its location in h is known.

Fredman and Tarjan [7] recently invented a heap implementation called the *Fibonacci heap* (abbreviated F-heap) that supports *delete min* and *delete* on an n-item heap in $O(\log n)$ amortized time and all the other heap operations listed above in $O(1)$ amortized time. (By *amortized time* we mean the time of an operation averaged over a worst-case sequence of operations. For a discussion of this concept see Tarjan's survey paper [19].) The importance of this result is in its reduction of the time for *decrease key* to $O(1)$ from the $O(\log n)$ of previous heap implementations. Since *decrease key* is a central operation in many network optimization algorithms, F-heaps lead to improved running times for such algorithms. In particular, Fredman and Tarjan reduced the running time of Dijkstra's shortest path algorithm from $O(m \log_{(m/n+2)} n)$ to $O(n \log n + m)$ and showed how to find minimum spanning trees in undirected graphs in $O(m\beta(m, n))$ time, improved from $O(m \log \log_{(m/n+2)} n)$. Here n and m are the numbers of vertices and edges in the problem graph, respectively, and $\beta(m, n) = \min\{i \mid \log^{(i)} n \leq m/n\}$, where $\log^{(i)} n$ is defined by $\log^{(0)}(x) = x$, $\log^{(i+1)} x = \log \log^{(i)} x$. (Throughout this paper we use base-two logarithms.)

Our purpose in this paper is to explore the use of F-heaps in computing minimum spanning trees in undirected and directed graphs. Our two main results, discussed in Sections 2 and 3, respectively, are as follows:

(i) By adding the idea of *packets* [8, 9] to the Fredman—Tarjan undirected minimum spanning tree algorithm, we reduce its running time to $O(m \log \beta(m, n))$ from $O(m\beta(m, n))$.

(ii) By observing that in certain situations items can be moved among F-heaps in $O(1)$ amortized time per item moved, we obtain an implementation of Edmonds' minimum directed spanning tree algorithm [16] with a running time of $O(n \log n + m)$, improved from the $O(n \log n + m \log \log \log_{(m/n+2)} n)$ bound of Gabow, Galil, and Spencer [8]. Our algorithm is also substantially simpler than the previous one.

Both our algorithms extend to allow a degree constraint at one vertex as we discuss in Section 4. Section 5 contains a few concluding remarks. The undirected minimum spanning tree algorithm described here originally appeared in a symposium paper [8] by the first three authors.

2. Minimum spanning trees in undirected graphs

Let $G = (V, E)$ be a connected, undirected graph with vertex set V of size n and edge set E of size m, such that each edge $\{v, w\}$ has a real-valued *cost* $c(v, w)$. Connectivity implies $m \geq n-1$; we shall assume $m \geq n$ since $m = n-1$ implies G itself is a tree. A *minimum spanning tree* of G is a spanning tree whose total edge cost is minimum. The problem of computing minimum spanning trees has a long history [11, 17]; the first algorithm was proposed by Boruvka [2] in 1926. Before the invention of F-heaps, the best known time bound was $O(m \log \log_{(m/n+2)} n)$ [4], a slight improvement over Yao's $O(m \log \log n)$ bound [20]. Fredman and Tarjan [7] used F-heaps to obtain an $O(m\beta(m, n))$ bound. We shall modify their algorithm so that it runs in $O(m \log \beta(m, n))$ time.

Our improvement uses the idea of *packets*. Packets can be used in a network optimization algorithm that makes a number of passes over the problem graph, during each of which it looks at all the edges but actually processes only a few of them. In such a situation we may be able to reduce the running time by reducing the apparent number of edges in the graph. We do this by grouping the edges into packets and handing only one edge per packet to the main algorithm. When the algorithm processes this edge we hand it the next edge in the packet.

To see how this idea applies to the computation of minimum spanning trees, let us examine the Fredman—Tarjan algorithm. It is a version of the following generalized greedy method. We maintain a forest defined by the edges so far selected to be in the minimum spanning tree. Initially this forest contains n one-vertex trees. We repeat the following step $n-1$ times (until there is only one n-vertex tree):
Connect. Select any tree T in the forest. Find a minimum-cost edge with exactly one endpoint in T and add it to the forest. (This connects two trees to form one.)

This algorithm is non-deterministic: we are free to select the tree to be processed in each connecting step in any way we wish. The Fredman—Tarjan version exploits this freedom by organizing the computation into *passes*. Each pass begins with a collection of *old trees* and combines them into a smaller collection of *new trees* by performing connecting steps. The new trees become the old trees for the next pass. After a sufficient number of passes only one tree remains, which is a minimum spanning tree.

In presenting the details of this method, we shall reformulate it to use packets. We replace each undirected edge $\{v, w\}$ by two directed edges (v, w) and (w, v), each of cost $c(v, w)$. The packet size p is a parameter of the algorithm. In addition, each pass of the algorithm has a parameter k, the *heap size*. At the beginning of a pass, for each old tree T, the remaining edges (v, w) such that v is in T are in packets, each of p or fewer edges. At most one packet for each tree T is designated as a *residual packet* containing $p/2$ or fewer edges. (The remaining edges are those that have not yet been added to the minimum spanning tree or discarded; if (v, w) is such an edge, v and w may or may not be in the same tree.) Each packet consists of a Fibonacci heap of its edges.

A pass consists of beginning with all trees marked *old* and repeating the following tree-expansion process until all trees are marked *new:*

Expand a tree:

1. Select any old tree T and mark it *current* (neither old nor new). Create a new, empty heap h. Heap h will contain old and new trees connected by an edge to T, each with an associated connecting edge of minimum cost. The cost of the connecting edge is the key of the tree in h. For each tree we maintain a bit indicating whether or not the tree is in h.

2. Create a set S containing all packets of T (those packets containing edges (v, w) with v in T). Set S will be used for updating h so that it contains trees connected to T, with minimum-cost connecting edges, including such a tree with globally minimum-cost connection.

3. Update h by repeating the following steps until S is empty (this selects one edge per packet to participate in the application of a connecting step to T):

3a. Remove any packet P from S. If P is empty, discard it. Otherwise, let (v, w) be a minimum-cost edge in P. (Vertex v is in T; vertex w may or may not be in T). Find the tree T' containing w.

3b. If $T'=T$ or if T' is in h with a connecting edge of cost not exceeding $c(v, w)$, delete (v, w) from P and put P back in S. Otherwise, proceed as follows. If T' is not in h use an *insert* operation to add T' to h with connecting edge (v, w) and key $c(v, w)$. If T' is in h with a connecting edge (x, y) such that $c(x, y)>c(v, w)$, use a *decrease key* operation to replace (x, y) as the connecting edge of T' by (v, w) and reduce the key of T' to $c(v, w)$; then delete (x, y) from the packet, say p', containing it and put p' in S.

4. If heap h is empty or has size exceeding k, mark T new, stop the tree expansion, and reset the bits indicating membership in h. Otherwise, apply a connecting step to T as follows. Perform a *delete min* operation on h, returning a tree T' with connecting edge (v, w). Add (v, w) to the minimum spanning tree, thereby combining T and T' into a single tree T''. If T and T' both have a residual packet, combine these packets using the *meld* operation. The melded packet becomes the residual packet of T'' if its size is no greater than $p/2$; otherwise T'' has no residual packet. If T' is new, associate the melded packet (if any) and all other packets of T and T' with T'' and stop the tree expansion. Otherwise (T' is old), add the melded packet (if any) and all other packets of T' to S. Delete (v, w) from the packet, say p, containing it and add p to S. Associate the melded packet (if any) and all other packets of T and T' with T''. Replace T by T'' and go to step 3 to update h.

The correctness of this algorithm follows from the observation that if the *delete min* operation in step 4 returns a tree T' with connecting edge (v, w), then (v, w) has minimum cost among all edges in all packets of T, since it is minimum among all packet minima. Thus the algorithm is a valid implementation of the generalized greedy method. Before analyzing its complexity, let us comment a little more on the data structures it requires.

We need a way, given a vertex, to find the tree containing it. The vertex sets of the trees are disjoint and updated by set union operations. Thus we can use a fast disjoint set union algorithm [14, 18] to maintain them. We must also maintain, for each tree, the set of its associated packets. These sets of packets are also combined by union, but there is no need to determine what tree is associated with a given packet. Thus we can represent each set of packets as a circular doubly-linked list, so that union of packet sets and deletion of a packet from a set take $O(1)$ time. The last important data structure is the heap h, which we implement as a Fibonacci heap.

We choose the packet size p to be $\beta(m, n)$. We initialize the packets by dividing the edges incident to each vertex into packets of size exactly p and at most one packet per vertex of size less than p; such a small packet is residual if its size is at most $p/2$. Initializing the packets as heaps takes $O(m)$ time since it requires $2m$ insertions in F-heaps. Each edge is deleted at most once from a packet, for a total deletion time of $O(m \log p)$. There are $n-1$ *meld* operations on packets, taking $O(n)$ time. Thus the overall time spent manipulating packets is $O(m \log \beta(m, n))$ plus $O(1)$ per packet minimum found.

Let k_i be the value of k selected for the i^{th} pass and let n_i be the number of trees at the beginning of the i^{th} pass. We choose $k_1=2^{2m/n}$ and $k_i=2^{k_{i-1}}$ for $i \geqq 2$. Let

us analyze the implications of this choice. Since a heap size bound of n or larger can only occur during the last pass, the number of passes is at most $\min\{i|\log^{(i)} n \leqq 2m/n\} \leqq \beta(m, n)$. The tree expansion stopping rule implies that at the end of the i^{th} pass each new tree has at least k_i associated packets. The number of packets existing at the beginning of pass i is at most $\min\{2m, 2m/\beta(m, n)+n_i\}$. Thus $n_{i+1} \leqq \min\{2m, 2m/2m/\beta(m, n)+n_i\}/k_i$. Since $m \geqq n$ and $2^x \geqq 2x$ for $x \geqq 2$, an induction on i implies $k_i \geqq 2^i(m/n)$ which combines with the estimate $n_{i+1} \leqq 2m/k_i$ to give $n_{i+1} \leqq n/2^{i-1}$. It follows that the number of packets existing at the beginning of the i^{th} pass is at most $2m/\beta(m, n)+n/2^{i-1}$, and $n_{i+1} \leqq (2m/\beta(m, n)+n/2^{i-1})/k_i$.

We can use these estimates to analyze the running time of the algorithm. The number of packet minima found during the algorithm is one per packet existing at the beginning of each pass, for a total of at most

$$\beta(m, n)\frac{2m}{\beta(m, n)}+\sum_{i=1}^{\beta(m,n)}\frac{n}{2^{i-1}} = 2(m+n) = O(m).$$

Consider the operations on heap h. There is one *insert* or *decrease key* per packet minimum found, for a total of $O(m)$ time over all passes. In addition, during pass i there are at most n_i-1 *delete min* operations, taking $O(n_i \log k_i)$ time. For pass 1 the *delete min* time is $O(n \log 2^{2m/n}) = O(m)$. For each subsequent pass the *delete min* time is $O(n_i k_{i-1}) = O(2m/\beta(m, n)+n/2^{i-2})$, Thus the *delete min* time summed over all passes is $O(m+n) = O(m)$.

The time for all other processing is $O(1)$ per packet minimum found, except for the operations of finding the tree containing a given vertex. There are $O(m)$ such operations, at most one per packet minimum found. The total time for the tree-finding operations is $O(m\alpha(m, n))$ [14, 18], where α is a functional inverse of Ackermann's function, which grows more slowly than $\log \beta$.

Combining all our estimates, we see that the overall running time of our minimum spanning tree algorithm is $O(m \log \beta(m, n))$.

3. Minimum spanning trees in directed graphs

Let $G=(V, E)$ be a directed graph with a distinguished root vertex r and a real-valued cost $c(v, w)$ on each edge (v, w). As in Section 2, we denote the number of vertices by n and the number of edges by m. We assume that every vertex of G is reachable from r. A *minimum spanning tree* of G is a spanning tree rooted at r (a set of $n-1$ edges containing paths from r to every vertex) of minimum total edge cost. Edmonds [6] devised a polynomial-time algorithm for finding a minimum spanning tree; the same method was discovered independently by Chu and Liu [5] and by Bock [1]. Edmonds' correctness proof uses concepts of linear programming; Karp [12] gave a purely combinatorial correctness proof. Tarjan [15] implemented Edmonds' algorithm to run in $O(\min\{m \log n, n^2\})$ time; Camerini et. al. [3] repaired an error in Tarjan's implementation. Gabow, Galil, and Spencer [8] devised a very complicated algorithm with a running time of $O(n \log n+m \log\log\log_{(m/n+2)} n)$. Here we shall develop a relatively simple algorithm with an $O(n \log n+m)$ time bound.

Let us begin by describing Edmonds' algorithm. To simplify matters we shall assume that G is strongly connected. (If not, we add dummy edges of suitably large cost from each vertex to r.) The algorithm consists of two phases. The first phase finds

a set of edges containing a minimum spanning tree and some additional edges. The second phase removes the extra edges.

The first phase begins with no edges selected and in general maintains a set of selected edges that defines a forest (set of trees). As the phase proceeds, cycles of selected edges are formed. Each such cycle is contracted to form a new (super-) vertex. At the end of the phase, all vertices in the graph are contracted into a single vertex. The phase consists of repeating the following step until only one vertex remains.

Grow. Choose any tree root v. Let (u, v) be a minimum-cost edge with $u \neq v$. Add (u, v) to the set of selected edges. If (u, v) forms a cycle with other selected edges, say (u, v), $(v=v_0, v_1)$, (v_1, v_2), ..., $(v_{k-1}, v_k=u)$, redefine the cost of each edge of the form (x, v_i) to be $c(x, v_i)-c(v_{(i-1) \bmod (k+1)}, v_i)$, and contract the set of vertices $\{v_0, v_1, \ldots, v_k\}$ to form a new vertex v^*. (Each edge with an endpoint in $\{v_0, v_1, \ldots, v_k\}$ has that endpoint replaced by v^*; loops and all but a minimum-cost edge among each set of multiple edges are deleted.)

Note. All edge cost changes in a growth step occur simultaneously. Adding an arbitrary constant to the cost of all edges entering a given vertex adds the same constant to the cost of all spanning trees; this does not affect the behavior of the algorithm.

The second phase of the algorithm consists of expanding the cycles formed during the first phase in reverse order of their contraction and discarding one edge from each to form a spanning tree in the original graph. Since the second phase of the algorithm is known to take $O(m)$ time [3, 15], we shall not discuss it further, except to note that by varying the expansion phase appropriately, we can find a minimum spanning tree with either specified or unspecified root.

Our improvement is in the first phase. We use a depth-first strategy to choose roots for growth steps. Specifically, we select an arbitrary vertex s and then repeatedly choose the root of the tree containing s for a growth step. This simplifies the implementation considerably, since the set of selected edges always forms a single path, which we call the *growth path*. We shall denote the vertices on the growth path by $v_0, v_1, \ldots, v_l$, with v_0 the root.

Let us fill in some details of the implementation. We mark every original vertex when it is connected to the growth path. This allows us to determine in $O(1)$ time whether a selected edge extends the growth path or forms a cycle. The total marking time is $O(n)$.

We represent the edges in the current contracted graph and their modified edge costs implicitly, using a *compressed tree* data structure [7, 14]. This data structure allows us to store a collection of disjoint sets, each initially a singleton and each element of which has an associated real value, subject to the following operations:

find(e): Return the name of the set containing element e.

find value(e): Return the current value of element e.

change value (Δ, A): Add Δ to the value of all elements in set A.

unite (A, B, C): Unite sets A and B, naming the union C. (This destroys the original sets A and B.)

In our application, the elements are the vertices of the original graph, and the sets are the sets of vertices contracted to form the vertices of the contracted graph. The name of a set is the corresponding vertex in the contracted graph. The value of an original vertex is the sum of the changes made to the cost of each of its incoming edges. That is, if (v, w) is an original edge and c is the original cost function, the corresponding current edge is $(\mathit{find}(v), \mathit{find}(w))$, of cost $c(v, w)+\mathit{find\ value}(w)$ (assuming that this edge has not yet been deleted).

When contracting a cycle containing vertices $v_0, v_1, \ldots, v_k$, we manipulate the compressed tree structure as follows. For $i=0, 1, \ldots, k$, let (x_i, y_i) be the original edge corresponding to the current edge $(v_{(i-1) \bmod (k+1)}, v_i)$. To update the edge costs, we perform *change value* $(-c(x_i, y_i)-\mathit{find\ value}(y_i), v_i)$ for each v_i, $i=0, 1, \ldots, k$. Then we use *unite* k times to combine the sets named $v_0, v_1, \ldots, v_k$ into a single set.

The various operations have the following running times. Any *change value* or *unite* operation takes $O(1)$ time. There are $O(n)$ such operations over the course of the algorithm. A total of k intermixed *find* and *find value* operations take $O((k+n)\alpha(k+n, n))$ time. As we shall see below, there are $O(n \log n+m)$ such operations over the course of the algorithm, which implies an $O(n \log n+m)$ time bound for all the *find* and *find value* operations, and hence for the total overhead for the compressed tree structure. (For $x \geqq n \log n$, $\alpha(x, n)=O(1)$ [14].)

Throughout the running of the algorithm, we represent each current edge by the corresponding original edge. Since we can easily obtain the current edge corresponding to any original edge, we shall ignore this distinction in what follows.

We need some lists and sets to keep track of candidate edges for selection in growth steps and to allow easy deletion of multiple edges. For each vertex v not on the growth path, we maintain an *exit list* of the current edges (v, v_i), sorted in increasing order on i. (Recall that the growth path contains the vertices $v_0, v_1, \ldots, v_l$.) For each vertex v_j on the growth path we maintain a similar exit list, of those edges (v_j, v_i) with $j>i$, also sorted in increasing order on i. Among the edges on an exit list, the first (of lowest index i) is *active;* the others are *passive*. For each vertex v_i, we maintain a *passive set* containing all passive edges entering v_i.

We manipulate the exit lists and passive sets as follows. Initially these lists and sets are all empty. After selecting a starting vertex s we add every edge of the form (v, s) to the exit list of v; each such edge is now active. After selecting a minimum-cost edge (u, v_0) during a growth step, we apply the appropriate one of the following two cases:

Case 1. u is not on the current growth path.

Vertex u becomes the new root of the growth path. We delete every edge from the exit set of u, removing each passive edge from the passive set containing it. (These edges, which lead to vertices on the growth path, are now unnecessary for the computation.) For each edge of the form (x, u), we add (x, u) to the front of the exit list of x. This edge is now active. If the list contains at least one other edge, the previously active edge, say (x, v_i), becomes passive; we add it to the passive set of v_i.

Case 2. u is on the current growth path, say $u=v_k$.

The cycle defined by the vertices $v_0, v_1, \ldots, v_k$ is contracted. We delete every edge from the passive list of every v_i, $1 \leqq i \leqq k$. For each such deleted edge (x, v_i), we delete from the exit list of x either (x, v_i) or the currently active edge, whichever

has larger cost, breaking a tie arbitrarily. (The contraction makes these edges multiple.) Each vertex among $v_1, v_2, \ldots, v_k$ now has at most one edge on its exit list, which we delete. The new vertex formed by the contraction has an empty exit list and an empty passive set.

If we represent the exit lists and passive sets by endogeneous* doubly linked lists, the total time for manipulating the exit lists and passive sets is $O(m)$. This is immediate from the following observations:

(i) Each edge is added to an exit list at most once. Any conversion of an edge from active to passive corresponds to such an addition.

(ii) Any deletion of an edge from a passive list causes the deletion of a (possibly different) edge from an exit list. Any conversion of an edge from passive to active corresponds to such a deletion.

We need one more data structure, to facilitate selection of minimum-cost edges during growth steps. For this purpose we use a collection of F-heaps. We shall describe a prototype algorithm which we then refine to achieve the desired time bound.

We define the *active set* of a vertex v_i on the growth path to be the set of vertices v such that (v, v_i) is an active edge. Since each vertex has at most one exiting active edge, the active sets are disjoint. For each vertex v_i on the growth path, we maintain an F-heap containing as items the vertices in the active set of v_i. The key of a vertex v in the heap is the current cost of the edge (v, v_i).

We manipulate the F-heaps as follows. To select a minimum-cost edge for use in a growth step, we perform *delete min* (v_0), where v_0 is the root of the growth path. If u is the vertex returned from the heap, (u, v_0) is the desired minimum-cost edge (also the first on the exit list of u). We handle the cases of extension of the growth path and contraction of a cycle as follows:

Case 1. u is not on the growth path.

We initialize a new, empty F-heap for u. For each edge of the form (x, u), if the exit list of x is empty, we insert x into the heap for u. Otherwise, let (x, v_i) be the currently active edge exiting x. We move x from the heap for v_i to the heap for u. (In either case we add (x, u) to the exit list of x as discussed above.) The key of x is now the cost of (x, u).

Case 2. u is on the current growth path, say $u = v_k$.

We update the heaps after first updating the costs of the edges entering $v_0, v_1, \ldots, v_k$ using the compressed tree data structure. When a currently active edge (x, v_i) is deleted from the exit set of a vertex x, we move x from the heap of v_i to the heap of the vertex v_j such that (x, v_j) is the newly active edge exiting x (note that $j \in \{0, 1, \ldots, k\}$); or, if there is no newly active edge, we merely delete x from the heap of v_i. Having completed all such updating, we meld the heaps for $v_0, v_1, \ldots, v_k$ to form a heap for the new vertex representing the contracted cycle.

---

* By an *endogenous* data structure we mean a linked data structure whose nodes are the items stored in the structure themselves. See Tarjan's monograph [17] for a discussion of this concept. Making the lists endogenous obviates the need for cross pointers between the occurrence of an edge on an exit list and its occurrence in a passive set.

To verify the correctness of this method, we need to know one fact about F-heaps. An F-heap consists of a set of rooted trees, whose nodes are the items in the heap. The items are arranged in *heap order:* for any node x and any child y of x, the key of x is no greater than the key of y. Consider the F-heap for a vertex v_j on the growth path. The key of a vertex v in the heap is the cost of the edge (v, v_j). Since any updating of edge costs changes the cost of two such edges by the same amount, such a change does not disturb the heap order. Thus we do not have to modify the F-heaps when updating edge costs. This implies that the implementation is correct.

To analyze the time spent manipulating F-heaps, we observe that there are at most n initializations of empty heaps, at most $2n-2$ *delete min* operations, at most n *insert* operations, at most $n-1$ *delete* operations, and at most $2m$ operations of moving a vertex from one heap to another. If we implement each move operation as a deletion followed by an insertion, the total time for heap operations is $O(m \log n)$. (To facilitate heap deletions, we make the heaps endogeneous. Alternatively, we can maintain a pointer from each vertex to its location in the heap containing it.)

By taking advantage of the special properties of F-heaps, we can reduce the amortized time per move operation from $O(\log n)$ to $O(1)$, thereby reducing the time for heap operations, and the overall time to compute a minimum directed spanning tree, from $O(m \log n)$ to $O(n \log n + m)$. This is the crucial idea in our improved algorithm.

Let us review the way F-heaps work. The two primitive operations on heap-ordered trees used in F-heaps are *linking*, in which we combine two trees into one by comparing their roots and making the root of smaller key the parent of the root larger key (breaking a tie arbitrarily), and *cutting*, in which we break one tree in two by deleting the edge joining a given node to its parent. Each cut or link operation takes $O(1)$ time. With each node in an F-heap we store its *rank*, defined to be the number of its children. In addition, each node is either *marked* or *unmarked*. When a root loses a comparison in a link operation and becomes a non-root, it becomes unmarked. We carry out the various operations on F-heaps as follows:*

make heap: Return a new, empty set of trees.

insert (x, h)*:* Make item x into a new, one-node tree, and add this tree to the forest representing h.

delete min (h)*:* In the forest of trees representing h, repeatedly link trees having roots of equal rank until no two roots have equal rank. Then choose any root of minimum key, delete it from its tree, and return it. (Each child of the deleted root becomes the root of a new tree in the forest.)

meld (h_1, h_2): Combine the forests representing h_1 and h_2 into a single forest.

decrease key (Δ, x, h): Subtract Δ from the key of item x. If x is not a tree root, cut the edge joining x to its parent, say y, and repeat the following step until it stops: If y is marked and not a root, make it unmarked, cut the edge joining y and its parent, let the new value of y be the parent, and continue; otherwise, mark y and stop.

---

* We have modified the data structure slightly to suit our needs. Specifically, we have dropped the maintenance of minimum nodes. (See [7],)

delete (x, h): Perform *decrease key* $(0, x, h)$. (This makes x a tree root without changing its key.) Delete x from the tree containing it. (Each child of x becomes a root.)

With this implementation, the maximum rank of a node is $O(\log n)$, where n is the total number of items. The amortized time per operation is $O(\log n)$ for *delete min* and *delete* and $O(1)$ for all the other operations. These bounds are derived by defining the potential of a collection of F-heaps to be the total number of trees plus twice the number of marked non-root nodes, and defining the amortized time of an operation to be its actual time plus the net increase in potential it causes. (For a discussion of the use of the "potential" concept in deriving amortized time bounds see [19].) For any sequence of heap operations starting with empty heaps, the total amortized time is an upper bound on the total actual time. For details of the analysis see [7].

We wish to implement a move operation, specified as follows:

move (x, h_1, h_2): Move item x from heap h_1 to heap h_2.

To carry out such an operation, we perform *decrease key* $(0, x, h_1)$, which makes x a tree root without changing its key. Then we move the entire tree rooted at x from heap h_1 to heap h_2.

The amortized time required by a move operation is $O(1)$, since moving a tree from one heap to another takes $O(1)$ actual time, does not change the potential, and does not affect the crucial structural property that the maximum rank of a node is $O(\log n)$.

Suppose we use this implementation of move operations in our directed minimum spanning tree algorithm. We certainly obtain an $O(n \log n + m)$ running time. The only question is whether the algorithm is correct. When a vertex v is moved from one heap to another, its key changes, because its active exiting edge changes. This may violate the heap order between v and its children in its heap. Furthermore, the descendants of v are now in the wrong heap. By making one more modification to the F-heap structure, we can fix these problems.

For any vertex v in the active set of some vertex v_i on the growth path, we call the heap of v_i the *home heap* of v, and we call v *displaced* if it is not on its home heap. When deleting a node from an F-heap (in a *delete min* or *delete* operation), we examine each child of the deleted node and, if the child is displaced, we move the tree rooted at the child to the home heap of the child. This does not affect the $O(\log n)$ amortized time of *delete min* or *delete*, since any node has $O(\log n)$ children and moving a tree takes $O(1)$ actual time and does not change the total potential.

This completes our description of the algorithm and the analysis of its running time. It remains for us to show that the manipulation of the F-heaps correctly produces minimum-cost edges for use in growth steps. The key to proving this is to observe that the manipulation of F-heaps maintains the following invariants:

(i) The root of any tree in an F-heap is always in its home heap (unless it has just become a root and is about to be moved to its home heap).

(ii) If x and y are vertices in an F-heap with x the parent of y, and v_i and v_j are the vertices on the growth path such that (x, v_i) and (y, v_j) are active, then $i \leqq j$.

(iii) If x and y are vertices in an F-heap with x the parent of y, and both x and y are in their home heap, then the key of x is no greater than the key of y.

Invariants (i) and (ii) are easy to prove by induction on the number of growth steps. (Invariant (ii) can become temporarily false during a contraction step (Case 2) but is restored by the melding that concludes the contraction.) To verify (iii), consider a link operation that makes a vertex x the parent of another vertex y. At the time of the link, both x and y are in their home heaps, say of vertex v_i, and the key of x is no greater than the key of y. As long as y remains a child of x, (y, v_i) remains active, and its cost remains the key of y. The only way to violate the heap order is to move x and its subtree to another heap, displacing y. Vertex y can subsequently return to its home heap (without deleting x) only if the entire part of the growth path up to and including the current vertex representing v_i is contracted. But this causes the key of x to decrease to at most the current cost of (x, v_i), thereby restoring the heap order between x and y. Invariant (iii) follows.

Let v_0 be the root of the current growth path. Invariants (i), (ii) and (iii) imply that any vertex of minimum key in the active set of v_0 must be a root of minimum key in the heap of v_0. This implies that the *delete min* operations used to find minimum-cost edges for growth steps work correctly. Hence the entire algorithm is correct.

4. Minimum spanning trees with a degree constraint

The algorithms described in Sections 2 and 3 extend to allow a degree constraint at one vertex. Let us first consider the undirected case. Suppose we wish to find a minimum spanning tree in an undirected graph among spanning trees with k edges incident to a specified vertex v, where k has a specified value. Gabow and Tarjan [10] have shown that this problem is linear-time equivalent to the unconstrained minimum spanning tree problem. Thus the algorithm of Section 2 combined with the Gabow—Tarjan reduction will find an undirected minimum spanning tree with a degree constraint in $O(m \log \beta(m, n))$ time.

The case of directed graphs requires a little more work. Suppose we wish to find a minimum spanning tree in a directed graph, with specified root r and exactly k edges exiting r for some given k. Gabow and Tarjan [10] describe an algorithm for solving this problem that runs in $O(m \log n)$ time. We shall provide a preprocessing step that reduces the number of edges that must be considered to $O(n)$, thus reducing the time for the actual spanning tree computation to $O(n \log n)$. The preprocessing step is a variant of the first phase of Edmonds' algorithm and takes $O(n \log n + m)$ time by the method of Section 3. Thus we obtain an $O(n \log n + m)$ time bound for finding a directed minimum spanning tree with a degree constraint.

We need a lemma that characterizes the effect of cycle contraction on directed minimum spanning trees. Let $G=(V, E)$ be a directed graph with edge cost function c, distinguished root vertex r, and having no edges entering r. Assume without loss of generality that no two edges entering the same vertex have the same cost. Let R be a set of required edges exiting r. An *R-tree* is a spanning tree rooted at r that contains all edges in R. Let $C=(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, v_0)$ be a cycle in G such that for $0 \leq i \leq k$, $(v_{(i-1) \bmod (k+1)}, v_i)$ is a minimum-cost edge entering v_i. Let G/C be the graph formed by changing the cost of each edge of the form (x, v_i) to $c(x, v_i) - c(v_{(i-1) \bmod (k+1)}, v_i)$, contracting the cycle C to a single vertex, and discarding all loops and all but a minimum-cost edge among each set of multiple edges. (If the set S of multiple edges contains an edge in R, the minimum-cost edge in $R \cap S$ is the

one retained.) Let R/C be the set of edges in G/C corresponding to R in G. Finally, let T be a minimum-cost R-tree in G, and let T' be a minimum-cost R/C-tree in G/C.

Lemma 1. *T is formed from T' by expanding the cycle C, adding all edges of R, and discarding every edge of C that enters a vertex with another entering edge.*

Proof. Let $U=\{(x, v_i)\in T\}$. Let $V=U\cap R$ if this set is non-empty, or otherwise let $V=\{(x, v_j)\}$, where (x, v_j) is an edge of U that is minimal in T in the sense of being closest to the root. Let $C'=\{(v_{i-1}, v_i)|$ no edge of V enters $v_i\}$.

We claim that $T-(U-V)\cup C'$ is a tree rooted at r containing all edges of R. Certainly $R\subseteq T-(U-V)\cup C'$, since $R\subseteq T$. Every vertex except r contains exactly one entering edge of $T-(U-V)\cup C'$. The only non-trivial part of the claim is to show that $T-(U-V)\cup C'$ contains no cycles. Suppose there is such a cycle, say Y. Y must contain an edge from a vertex not on C to a vertex on C; this edge cannot be in R since r has no entering edges. It follows that Y can only exist if $V=\{(x, v_j)\}$ where (x, v_j) is an edge of U that is minimal in T and Y contains (x, v_j). But the path in T from r to x does not contain a vertex of C by the choice of (x, v_j), which implies that (x, v_j) cannot be on a cycle in $T-(U-V)\cup C'$. This contradiction establishes the claim.

Since each edge $(v_{i-1}, v_i)\in C'$ is of minimum cost among edges entering v_i, we must have $U-V=C'$, since otherwise $T-(U-V)\cup C'$ has cost less than T, contradicting the fact that T is a minimum-cost R-tree. Thus either the edges in T entering vertices on C are exactly the edges in R entering vertices on C (if $U\cap R\neq\emptyset$), or there is exactly one edge in T entering a vertex on C (if $U\cap R=\emptyset$). In either case T is formed from *some* R/C-tree T'' in G/C by expanding C, adding all edges of R, and discarding every edge of C that enters a vertex with another entering edge.

Suppose $U\cap R=\emptyset$. Then exactly one edge of C is discarded in forming T from T''. The edge cost transformation is such that $c(T)=c'(T'')+c(C)$. Since T is minimum with respect to c, T'' must be minimum with respect to c', and $T''=T'$ since G/C has a unique minimum-cost R/C-tree.

A similar argument applies if $U\cap R\neq\emptyset$. In this case the edge of T'' entering the vertex formed by contracting C corresponds to the minimum-cost edge in R entering a vertex of C. It follows that $c(T)$ is a non-decreasing function of $c'(T'')$ (or of $c(T'')$; in this case the edge cost transformation is irrelevant). Since T is minimum-cost, T'' must also be minimum-cost, and $T''=T'$. ∎

Lemma 1 implies that the desired degree-constrained minimum spanning tree T is contained in a subgraph H constructed as follows. Choose any vertex $r'\neq r$. Form $G-r$; add an edge (r', v) of appropriately high cost to any vertex v not reachable from r'. Run phase 1 of Edmonds' algorithm on this graph. Then H consists of all edges in G corresponding to edges selected in phase 1, plus all edges directed from r. To prove that T is contained in H, observe that H contains all cycles contracted in phase 1. Now use induction on the number of cycles contracted in phase 1, taking R as the set of edges incident to r in T.

The algorithm to find a minimum degree-constrained spanning tree constructs H using the algorithm of Section 3 and runs the Gabow—Tarjan algorithm on H. The time is $O(m+n\log n)$, since H has $O(n)$ edges.

We close this section by noting that Lemma 1, specialized to the case $R=\emptyset$, provides the basis for a purely combinatorial proof of correctness of Edmonds' algorithm that is somewhat simpler than Karp's proof [12].

5. Remarks

We have described an $O(m \log \beta(m, n))$-time algorithm for finding undirected minimum spanning trees and an $O(n \log n + m)$-time algorithm for finding directed minimum spanning trees. Both algorithms extend to allow a degree constraint at one vertex (the root, in the directed case). The algorithms use F-heaps and some additional techniques.

Remaining open questions have to do with whether our algorithms are improvable and whether the methods they use can be applied to other problems. In this regard, it is known that undirected minimum spanning trees can be verified in $O(m\alpha(m, n))$ time [16] and even in $O(m)$ comparisons [13]. Thus one is tempted to look for an $O(m\alpha(m, n))$-time or even $O(m)$-time algorithm for finding such trees. On the other hand, Edmonds' directed minimum spanning tree algorithm can be used to sort n numbers. This implies an $\Omega(n \log n + m)$ lower bound for any implementation of his algorithm that makes only binary decisions. Any improvement would require a new algorithmic approach rather than just new implementation ideas.

The most intriguing problem to which the techniques discussed in this paper might be applicable is nonbipartite weighted matching. The current best time bound is $O(n^2 \log n + nm \log \log \log_{(m/n+2)} n)$, obtained by Gabow, Galil, and Spencer [8, 9]. We might hope for an improvement to $O(n^2 \log n + nm)$.

Acknowledgement. The fourth author would like to thank Ilan Bar-On for assistance in working out the details of the directed minimum spanning tree algorithm.

References

[1] F. Bock, An algorithm to construct a minimum directed spanning tree in a directed network, *in: Developments in Operations Research*, Gordon and Breach, New York, 1971, 29—44.
[2] O. Boruvka, O jistem problemu minimální'm *Práca Moravské Prírodovědecké Společnosti* **3** (1926), 37—58. (In Czech).
[3] P. M. Camerini, L. Fratta and F. Maffioli, A note on finding optimum branchings, *Networks* **9** (1979), 309—312.
[4] D. Cheriton and R. E. Tarjan, Finding minimum spanning trees, *SIAM J. Comput.* **5** (1976), 724—742.
[5] Y. J. Chu and T. H. Liu, On the shortest arborescence of a directed graph, *Sci. Sinica* **14** (1965), 1396—1400.
[6] J. Edmonds, Optimum branchings, *J. Res. Nat. Bur. Standards* **71B** (1967), 233—240.
[7] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in network optimization algorithms, *J. Assoc. Comput. Mach., to appear; also Proc. 25th Annual IEEE Symp. on Found. of Comp. Sci.* (1984), 338—346.
[8] H. N. Gabow, Z. Galil and T. H. Spencer, Efficient implementation of graph algorithms using contraction, *Proc. 25th Annual IEEE Symp. on Found. of Comp. Sci.* (1984), 347—357.
[9] H. N. Gabow, Z. Galil and T. Spencer, Efficient implementation of graph algorithms using contraction, *J. Assoc. Comput. Mach., submitted.*
[10] H. N. Gabow and R. E. Tarjan, Efficient algorithms for a family of matroid intersection problems, *J. Algorithms* **5** (1984), 80—131.
[11] R. L. Graham and P. Hell, On the history of the minimum spanning tree problem, *Ann. History of Computing* **7** (1985), 43—57.
[12] R. M. Karp, A simple derivation of Edmonds' algorithm for optimum branchings, *Networks* **1** (1971), 265—272.
[13] J. Komlós, Linear verification for spanning trees, *Combinatorica* **5** (1985), 57—65.
[14] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* **22** (1975), 215—225.

[15] R. E. TARJAN, Finding optimum branchings, *Networks* **7** (1977), 25—35.
[16] R. E. TARJAN, Applications of path compression on balanced trees, *J. Assoc. Comput. Mach.* **26** (1979), 690—715.
[17] R. E. TARJAN, *Data Structures and Network Algorithms*, CBMS 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
[18] R. E. TARJAN and J. VAN LEEUWEN, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.* **31** (1984), 245—281.
[19] R. E. TARJAN, Amortized computational complexity, *SIAM J. Alg. Disc. Meth.* **6** (1985),306—318.
[20] A. YAO, An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees, *Inform. Process. Lett.* **4** (1975), 21—23.

Harold N. Gabow
University of Colorado
Boulder, CO 80309
U.S.A.

Zvi Galil
Columbia University
New York, NY 10027
U.S.A.

and

Tel Aviv University
Tel Aviv, Israel

Thomas Spencer
Rensselaer Polytechnic Inst.
Troy, NY 12 180
U.S.A.

Robert E. Tarjan
AT&T Bell Laboratories
Murray Hill, NJ 07974
U.S.A.