FIBONACCI HEAPS AND THEIR USES IN IMPROVED NETWORK OPTIMIZATION ALGORITHMS

Michael L. Fredman

EECS Department
University of CA, San Diego
La Jolla, California 92093

Robert Endre Tarjan

AT&T Bell Laboratories
Rm. 2C-362, 600 Mountain Ave.
Murray Hill, New Jersey 07974

ABSTRACT

In this paper we develop a new data structure for implementing heaps (priority queues). Our structure, Fibonacci heaps (abbreviated F-heaps), extends the binomial queues proposed by Vuillemin and studied further by Brown. F-heaps support arbitrary deletion from an n-item heap in O(log n) amortized time and all other standard heap operations in O(1) amortized time. Using F-heaps we are able to obtain improved running times for several network optimization algorithms.

1. Introduction

A heap is an abstract data structure consisting of a set of items, each with a real-valued key, subject to the following operations:

make heap: Return a new, empty heap.

insert(i,h): Insert a new item i with predefined key into heap h.

find min(h): Return an item of minimum key in heap h. This operation does not change h.

delete min(h): Delete an item of minimum key from h and return it.

In addition, the following operations on heaps are often useful:

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

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

delete(i,h): Delete arbitrary item i from heap h. This operation assumes that the position of i in h is known.

Note. Other authors have used different terminology for heaps. Knuth [14] called heaps priority queues. Aho, Hopcroft, and Ullman [1] used this term for heaps not subject to melding and called meldable heaps mergeable heaps. □

In our discussion of heaps we shall assume that a given item is in only one heap at a time and that a pointer to its heap position is maintained. It is important to remember that heaps do not support efficient searching for an item.

Vuillemin [27] invented a class of heaps, called binomial queues, that support all the heap operations in O(log n) worst-case time. Here n is the number of items in the heap or heaps involved in the operation. Brown [2] studied alternative representations of binomial heaps and developed both theoretical and experimental running time bounds. His results suggest that binomial queues are a good choice in practice if meldable heaps are needed, although several other heap implementations have the same O(log n) worst-case time bound. For further discussion of heaps, see [1,2,14,24].

In this paper we develop an extension of binomial queues called Fibonacci heaps, abbreviated F-heaps. F-heaps support delete min and delete in O(log n) amortized time and all the other heap operations, in particular decrease key, in O(1) amortized time. For situations in which the number of deletions is small compared to the total number of operations, F-heaps are asymptotically faster than binomial queues.

Heaps have a variety of applications in network optimization, and in many such applications the number of deletions is relatively small. Thus we are able to use F-heaps to obtain asymptotically faster algorithms for several well-known network optimization problems. Our original purpose in developing F-heaps was to speed up Dijkstra's algorithm for the single-source shortest path problem with non-negative length edges [5]. Our implementation of Dijkstra's algorithm runs in $O(n \log n + m)$ time, improved from Johnson's $O(m \log_{(m/n+2)} n)$ bound [12,24].

0272-5428/84/0000/0338$01.00 © 1984 IEEE

Various other network algorithms use Dijkstra's algorithm as a subroutine, and for each of these we obtain a corresponding improvement. Thus we can solve both the all-pairs shortest path problem with possibly negative length edges and the assignment problem (weighted bipartite matching) in $O(n^2 \log n + nm)$ time, improved from $O(nm \log_{(m/n+2)} n)$ [24].

We also obtain a faster method for computing minimum spanning trees. Our bound is $O(m\beta(m,n))$, improved from $O(m \log \log_{(m/n+2)} n)$ [4,24], where $\beta(m,n) = \min\{i \mid \log^{(i)} n \leq m/n\}$.

All our bounds for network optimization are asymptotic improvements for graphs of intermediate density ($n << m << n^2$). Our bound for minimum spanning trees, which is perhaps our most striking result, is an asymptotic improvement for sparse graphs as well.

The remainder of the paper consists of four sections. In Section 2 we develop and analyze F-heaps. In Section 3 we use F-heaps to implement Dijkstra's algorithm. In Section 4 we discuss the minimum spanning tree problem. In Section 5 we mention several more recent results and remaining open problems.

2. Fibonacci Heaps

To implement heaps we use heap-ordered trees. A heap-ordered tree is a rooted tree containing a set of items, one item in each node, with the items arranged in heap order: if x is any node, then the key of the item in x is no less than the key of the item in its parent p(x), provided x has a parent. Thus the tree root contains an item of minimum key. The fundamental operation on heap-ordered trees is linking, which combines two item-disjoint trees into one. Given two trees with roots x and y, we link them by comparing the keys of the items in x and y. If the item in x has smaller key, we make y a child of x; otherwise, we make x a child of y. (See Figure 1.)

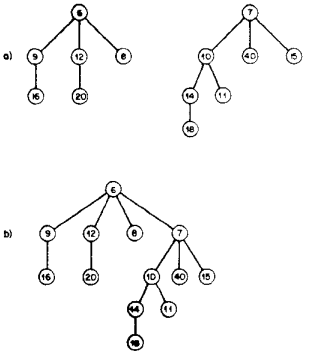

Figure 1. Linking two heap-ordered trees. (In this and most later figures, we do not distinguish between items and their keys.)
(a) Two trees.
(b) After linking.

A Fibonacci heap (or F-heap) is a collection of item-disjoint heap-ordered trees. We impose no explicit constraints on the number or structure of the trees; the only constraints are implicit in the way the trees are manipulated. We call the number of children of a node x its rank r(x). There is no constant upper bound on the rank of a node, although we shall see that the rank of a node with n descendants is O(log n). Each node is either marked or unmarked; we shall discuss the use of marking later.

In order to make the correctness of our claimed time bounds obvious, we shall assume the following representation of F-heaps. Each node contains a pointer to its parent (or to a special node null if it has no parent) and a pointer to one of its children. The children of each node are doubly linked in a circular list. Each node also contains its rank and a bit indicating whether it is marked. The roots of all the trees in the heap are doubly linked in a circular list. We access the heap by a pointer to a root containing an item of minimum key; we call this root the minimum node of the heap. A minimum node of null denotes an empty heap. (See Figure 2.) This representation requires space in each node for one item, four pointers, an integer, and a bit. The double linking of the lists of roots and children makes deletion from such a list possible in O(1) time. The circular linking makes concatenation of lists possible in O(1) time.

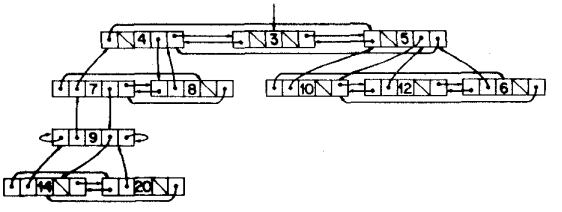

Figure 2. Pointers representing an F-heap. The four pointers in each node indicate the left sibling, parent, some child, and right sibling. The middle field in each node is its key. Ranks and mark bits are not shown.

We shall postpone a discussion of decrease key and delete until later in the section. The remaining heap operations we carry out as follows. To perform make heap, we return a pointer to null. To perform find min(h), we return the item in the minimum node of h. To carry out insert(i,h), we create a new heap consisting of one node containing i and replace h by the meld of h and the new heap. To carry out meld(h_1,h_2), we combine the root lists of h_1 and h_2 into a single list and return as the minimum node of the new heap either the minimum node of h_1 or the minimum node of h_2, whichever contains the item of smaller key. (In the case of a tie, the choice is arbitrary.) All of these operations take O(1) time in the worst case.

The most time-consuming operation is delete min(h). We begin the deletion by removing the minimum node, say x, from h. Then we concatenate

the list of children of x with the list of roots of h other than x and repeat the following step until it no longer applies.

Linking Step. Find any two trees whose roots have the same rank and link them. (The new tree root has rank one greater than the ranks of the old tree roots.)

Once there are no two trees with roots of the same rank, we form a list of the remaining roots, in the process finding a root containing an item of minimum key to serve as the minimum node of the modified heap. We complete the deletion by saving the item in x, destroying x, and returning the saved item. (See Figure 3.)

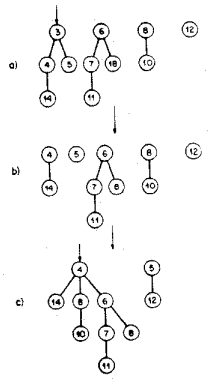

Figure 3. Effect of a delete min operation.
(a) Before the deletion.
(b) After removal of the minimum node, containing 3.
(c) After linking 4 and 8, then 4 and 6, then 5 and 12.

The delete min operation requires finding pairs of tree roots of the same rank to link. To do this we use an array indexed by rank, from one up to the maximum possible rank. Each array position holds a pointer to a tree root. When performing a delete min operation, after deleting the minimum node and forming a list of the new tree roots, we insert the roots one-by-one into the appropriate array positions. Whenever we attempt to insert a root into an already-occupied position, we perform a linking step and attempt to insert the root of the new tree into the next higher position. After successfully inserting all the roots, we scan the array, emptying it. The total time for the delete min operation is proportional to the maximum rank of any of the nodes manipulated plus the number of linking steps.

The data structure we have so far described is a "lazy melding" version of binomial queues. If we begin with no heaps and carry out an arbitrary sequence of heap operations (not including delete or decrease key), then each tree ever created is a binomial tree, defined inductively as follows: a binomial tree of rank zero consists of a single node; a binomial tree of rank $k > 0$ is formed by linking two binomial trees of rank $k - 1$. (See Figure 4.) A binomial tree of rank k contains exactly 2^k nodes, and its root has exactly k children. Thus every node in an n-item heap has rank at most log n.†

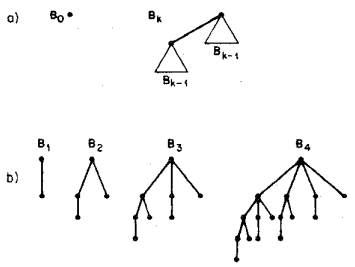

Figure 4. Binomial trees.
(a) Inductive definition.
(b) Examples.

We can analyze the amortized running times of the heap operations by using the "potential" technique of Sleator and Tarjan [19,25]. We assign to each possible collection of heaps a real number called the potential of the heaps. We define the amortized time of a heap operation to be its actual running time plus the net increase it causes in the potential. (A decrease in potential counts negatively and thus makes the amortized time less than the actual time.) With this definition, the actual time of a sequence of operations is equal to the total amortized time plus the net decrease in potential over the entire sequence.

To apply this technique, we define the potential of a collection of heaps to be the total number of trees they contain. If we begin with no heaps, the initial potential is zero and the potential is always non-negative. Thus the total amortized time of a sequence of operations is an upper bound on the total actual time. The amortized time of a make heap, find min, insert, or meld operation is $O(1)$: an insertion increases the number of trees by one; the other operations do not affect the number of trees. If we charge one unit of time for each linking step, then a delete min operation has an amortized time of $O(\log n)$, where n is the number of items in the heap: deleting the minimum node increases the number of trees by at most log n; each linking step decreases the number of trees by one.

Our goal now is to extend this data structure and its analysis to include the remaining heap operations. We implement decrease key and delete as follows. To carry out decrease key(Δ,i,h), we subtract Δ from the key of i, find the node x containing i, and cut the edge joining x to its parent p(x). This causes the subtree rooted at x to become a new tree of h, and requires decreasing the rank of p(x) and adding x to the list of roots of h. (See Figure 5.) (If x is originally a root, we carry out decrease key(Δ,i,h) merely by subtracting Δ from the key of i.) If the new key of

† All logarithms in this paper for which a base is not explicitly specified are base two.

i is smaller than the key of the minimum node, we redefine the minimum node to be x. This method works because Δ is non-negative; decreasing the key of i preserves heap order within the subtree rooted at x, though it may violate heap order between x and its parent. A decrease key operation takes O(1) actual time.

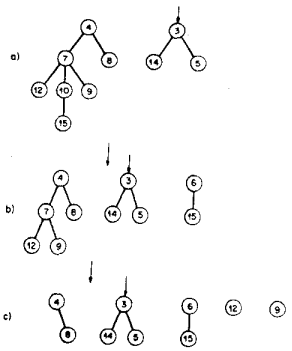

Figure 5. The decrease key and delete operations.
(a) Original heap.
(b) After reducing key 10 to 6. The minimum node is still the node containing 3.
(c) After deleting key 7.

The delete operation is similar to decrease key. To carry out delete(i,h), we find the node x containing i, cut the edge joining x and its parent, form a new list of roots by concatenating the list of children of x with the original list of roots, and destroy node x. (See Figure 5.) (If x is originally a root, we remove it from the list of roots rather than removing it from the list of children of its parent; if x is the minimum node of the heap, we proceed as in delete min.) A delete operation takes O(1) actual time, unless the node destroyed is the minimum node.

There is one additional detail of the implementation that is necessary to obtain the desired time bounds. After a root node x has been made a child of another node by a linking step, as soon as x loses two of its children through cuts, we cut the edge joining x and its parent as well, making x a new root as in decrease key. We call such a cut a cascading cut. A single decrease key or delete operation in the middle of a sequence of operations can cause a possibly large number of cascading cuts. (See Figure 6.)

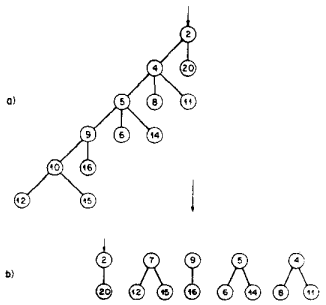

Figure 6. Cascading cuts.
(a) Just before decreasing the key of 10. Nodes 4,5,9 are assumed to have previously lost a child via a cut.
(b) After decreasing the key of 10 to 7. Original cut separates 10 from 9. Cascading cuts separate 9 from 5, 5 from 4, and 4 from 2.

The purpose of marking nodes is to keep track of where to make cascading cuts. When making a root node x a child of another node in a linking step we unmark x. When cutting the edge joining a node x and its parent p(x), we decrease the rank of p(x) and check whether p(x) is a root. If p(x) is not a root, we mark it if it is unmarked and cut the edge to its parent if it is marked. (The latter case may lead to further cascading cuts.) With this method, each cut takes O(1) time.

This completes the description of F-heaps. Our analysis of F-heaps hinges on two crucial properties: (i) Each tree in an F-heap, even though not necessarily a binomial tree, has size at least exponential in the rank of its root; (ii) The number of cascading cuts that takes place during a sequence of heap operations is bounded by the number of decrease key and delete operations. Before proving these properties, we remark that cascading cuts are introduced in the manipulation of F-heaps for the purpose of preserving property (i). Moreover, the condition for their occurrence, namely the "loss of two children" rule, limits the frequency of cascading cuts as described by (ii). The following lemma implies (i):

Lemma 1. Let x be any node in an F-heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the ith child of x has rank at least i - 2.

Proof. Let y be the ith child of x and consider the time when y was linked to x. Just before linking, x had at least i - 1 children (some of which it may have lost after the linking). Since x and y had the same rank just before the linking, they both had rank at least i - 1 at this time. After the linking, the rank of y could have decreased by at most one without causing y to be cut as a child of x. □

Corollary 1. A node of rank k in an F-heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself, where F_k is the kth Fibonacci number ($F_0 = 0$, $F_1 = 1, F_k = F_{k-2} + F_{k-1}$ for $k \geq 2$), and $\phi = (1 + \sqrt{5})/2$ is the golden ratio. (See Figure 7.)

Proof. Let S_k be the minimum possible number of descendants of a node of rank k. Obviously, $S_0 = 1$, $S_1 = 2$. Lemma 1 implies that $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ for $k \geq 2$. The Fibonacci numbers satisfy $F_{k+2} = \sum_{i=2}^{k} F_i + 2$, from which $S_k \geq F_{k+2}$ for $k \geq 0$ follows by induction on k. The inequality $F_{k+2} \geq \phi^k$ is well-known [11]. □

Remark. This corollary is the source of the name "Fibonacci heap." □

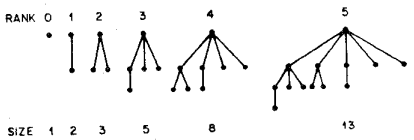

Figure 7. Trees of minimum possible size for a given rank in a Fibonacci heap.

To analyze F-heaps, we need to extend our definition of potential. We define the potential of a collection of F-heaps to be the number of trees plus twice the number of marked non-root nodes. The O(1) amortized time bounds for make heap, find min, insert, and meld remain valid, as does the O(log n) bound for delete min; delete min(h) increases the potential by at most 1.4404 log n minus the number of linking steps, since if the minimum node has rank k, then $\phi^k \leq n$, and thus $k \leq \log n / \log \phi \leq 1.4404 \log n$.

Let us charge one unit of time for each cut. A decrease key operation causes the potential to increase by at most three minus the number of cascading cuts, since the first cut converts a possibly unmarked non-root node into a root, each cascading cut converts a marked non-root node into a root, and the last cut (either first or cascading) can convert a non-root node from unmarked to marked. If follows that decrease key has an O(1) amortized time bound. Combining the analysis of decrease key with that of delete min, we obtain an O(log n) amortized time bound for delete. Thus we have the following theorem:

Theorem 1. If we begin with no F-heaps and perform an arbitrary sequence of F-heap operations, then the total time is at most the total amortized time, where the amortized time is O(log n) for each delete min or delete operation and O(1) for each of the other operations.

We close this section with a few remarks on the storage utilization of F-heaps. Our implementation of F-heaps uses four pointers per node, but this can be reduced to three per node by using an appropriate alternative representation [2] and even to two per node by using a more complicated representation, at a cost of a constant factor in running time. Although our implementation uses an array for finding roots of the same rank to link, random-access memory is not actually necessary for this purpose. Instead, we can maintain a doubly linked list of rank nodes representing the possible ranks. Each node has a rank pointer to the rank node representing its rank. Since the rank of a node is initially zero and only increases or decreases by one, it is easy to maintain the rank pointers. When we need to carry out linking steps, we can use each rank node to hold a pointer to a root of the appropriate rank. Thus the entire data structure can be implemented on a pointer machine [22] with no loss in asymptotic efficiency.

3. Shortest Paths

In this section we use F-heaps to implement Dijkstra's shortest path algorithm [5] and explore some of the consequences. Our discussion of Dijkstra's algorithm is based on Tarjan's presentation [24]. Let G be a directed graph, one of whose vertices is distinguished as the source s, and each of whose edges (v,w) has a non-negative length $\ell(v,w)$. The length of a path in G is the sum of its edge lengths; the distance from a vertex v to a vertex w is the minimum length of a path from v to w. A minimum-length path from v to w is called a shortest path. The single-source shortest path problem is that of finding a shortest path from s to v for every vertex v in G.

We shall denote the number of vertices G by n and the number of edges by m. We assume that there is a path from s to any other vertex; thus $m \geq n-1$. Dijkstra's algorithm solves the shortest path problem using a tentative distance function d from vertices to real numbers with the following properties:

(i) for any vertex v such that d(v) is finite, there is a path from s to v of length d(v);

(ii) when the algorithm terminates, d(v) is the distance from s to v.

Initially $d(s) = 0$ and $d(v) = \infty$ for $v \neq s$. During the running of the algorithm, each vertex is in one of three states: unlabeled, labeled, or scanned. Initially s is labeled and all other vertices are unlabeled. The algorithm consists of repeating the following step until there are no labeled vertices (every vertex is scanned):

Scan. Select a labeled vertex v with d(v) minimum. Convert v from labeled to scanned. For each edge (v,w) such that $d(v) + \ell(v,w) < d(w)$, replace d(w) by $d(v) + \ell(v,w)$ and make w labeled.

The non-negativity of the edge lengths implies that a vertex, once scanned, can never become labeled, and further that the algorithm computes distances correctly.

To implement Dijkstra's algorithm, we use a heap to store the set of labeled vertices. The tentative distance of a vertex is its key. Initialization requires one make heap and one insert operation. Each scanning step requires one delete min operation. In addition, for each edge (v,w) such that $d(v) + \ell(v,w) < d(w)$, we need either an insert or a decrease key. There is one make heap operation, n insert and n delete min operations, and at most m decrease key operations. The maximum heap size is n - 1. If we use an F-heap, the total time for heap operations is $O(n \log n + m)$. The time for other tasks is $O(n+m)$. Thus we obtain an $O(n \log n + m)$ running time for Dijkstra's algorithm. This improves Johnson's bound of $O(m \log_{(m/n+2)} n)$ [12,24] based on the use of implicit heaps with a branching factor of m/n+2.

By augmenting the algorithm we can make it compute shortest paths as well as distances: for each vertex v, we maintain a tentative predecessor pred(v) that immediately precedes v on a tentative shortest path from s to v. When replacing d(w) by $d(v) + \ell(v,w)$ in a scanning step, we define pred(w) = v. Once the algorithm terminates, we can find a shortest path from s to v for any vertex v by following predecessor pointers from v back to s. Augmenting the algorithm to maintain predecessors adds only O(m) to the running time.

Several other optimization algorithms use Dijkstra's algorithm as a subroutine, and for each of these we obtain a corresponding improvement. We shall give three examples.

(i) The all-pairs shortest path problem, with or without negative edge lengths, can be solved in O(nm) time plus n iterations of Dijkstra's algorithm [12,23]. Thus we obtain a time bound of $O(n^2 \log n + nm)$, improved from $O(nm \log_{(m/n+2)} n)$.

(ii) The assignment problem (bipartive weighted matching) can also be solved in O(nm) time plus n iterations of Dijkstra's algorithm [24]. Thus we obtain a bound of $O(n^2 \log n + nm)$, improved from $O(nm \log_{(m/n+2)} n)$.

(iii) Knuth's generalization [15] of Dijkstra's algorithm to compute minimum-cost derivations in a context-free language runs in $O(n \log n + t)$ time, improved from $O(m \log n + t)$, where n is the number of non-terminals, m is the number of productions, and t is the total length of the productions. (There is at least one production per non-terminal; thus $m \geq n$.)

4. Minimum Spanning Trees

A less immediate application of F-heaps is to the computation of minimum spanning trees. See [24] for a systematic discussion of minimum spanning tree algorithms. Let G be a connected undirected graph with n vertices and m edges (v,w) each of which has a non-negative cost c(v,w). A minimum spanning tree of G is a spanning tree of G of minimum total edge cost.

We can find a minimum spanning tree by using a generalized greedy approach. We maintain a forest defined by the edges so far selected to be in the minimum spanning tree. We initialize the forest to contain each of the n vertices of G as a one-vertex tree. Then 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. One possibility is to always select the tree containing a fixed start vertex s, thereby growing a single tree T that absorbs the vertices of G one-by-one. This algorithm, usually credited to Prim [17] and Dijkstra [5] independently, was actually discovered by Jarnik [11] (see Graham and Hell's survey paper [10]). The algorithm is almost identical to Dijkstra's shortest path algorithm, and we can implement it in the same way. For each vertex v not yet in T, we maintain a key measuring the tentative cost of connecting v to T. If $v \neq s$ and $key(v) < \infty$, we also maintain an edge e(v) by which the connection can be made. We start by defining $key(s) = 0$ and $key(v) = \infty$ for $v \neq s$. Then we repeat the following step until no vertex has finite key:

Connect to start. Select a vertex v with key(v) minimum among vertices with finite key. Replace key(v) by $-\infty$. For each edge (v,w) such that $c(v,w) < key(w)$, replace key(w) by c(v,w) and define e(w) = (v,w).

When this algorithm terminates, the set of edges e(v) with $v \neq s$ defines a minimum spanning tree. The purpose of setting $key(v) = -\infty$ in the connecting step is to mark v as being in T. If we store the vertices with finite key in an F-heap the algorithm requires n delete min operations and O(m) other heap operations, none of them deletions. The total running time is $O(n \log n + m)$.

The best previous bound for computing minimum spanning trees is $O(m \log \log_{(m/n+2)} n)$ [4,24], a slight improvement over Yao's $O(m \log \log n)$ bound [28]. Our $O(n \log n + m)$ bound is better then the old bound for graphs of intermediate density (e.g. $m = \Theta(n \log n)$). However, we can do better.

The idea is to grow a single tree only until its heap of neighboring vertices exceeds a certain critical size. Then we start from a new vertex and grow another tree, again stopping when the heap gets too large. We continue in this way until every vertex is in a tree. Then we condense every tree into a single super-vertex and begin a new pass of the same kind over the condensed graph. After a sufficient number of passes, only one super-vertex will remain, and by expanding the super-vertices we can extract a minimum spanning tree.

Our implementation of this method does the condensing implicitly. We shall describe a single pass of the algorithm. We begin with a forest of previously grown trees, which we call old trees, defined by the edges so far added to the forest. The pass connects these old trees into new larger trees that become the old trees for the next pass.

To start the pass, we number the old trees consecutively from one and assign to each vertex the number of the tree containing it. This allows us to refer to trees by number and to directly access the old tree tree(v) containing any vertex v. Next, we do a cleanup, which takes the place of condensing. We discard every edge connecting two vertices in the same old tree and all but a minimum-cost edge connecting each pair of old trees. We can do such a cleanup in O(m) time by sorting the edges lexicographically on the numbers of the trees containing their endpoints, using a two-pass radix sort. Then we scan the sorted edge list, saving only the appropriate edges. After the cleanup, we construct a list for each old tree T of the edges with one endpoint in T. (Each edge will appear in two such lists.)

To grow new trees, we give every old tree a key of ∞ and unmark it. We create an empty heap. Then we repeat the following tree-growing step until there are no unmarked old trees. This completes the pass.

Grow a New Tree. Select any unmarked old tree T_0 and insert it as an item into the heap with a key of zero. Repeat the following step until the heap is empty, or it contains more than k trees (where k is a parameter to be chosen later), or the growing tree becomes connected to a marked old tree:

> Connect to Starting Tree. Delete an old T of minimum key from the heap. Set key(T) = $-\infty$. If $T \neq T_0$ (i.e. T is not the starting tree of this growth step), add e(T) to the forest. If T is marked, stop growing the current tree and finish the growth step as described below. Otherwise mark T. For each edge (v,w) with v in T and c(v,w) < key(tree(w)), set e(tree(w)) = (v,w). If key(tree(w)) = ∞, insert tree(w) in the heap with a redefined key of c(v,w); if c(v,w) < key(tree(w)) < ∞, decrease the key of tree(w) to c(v,w).

To finish the growth step, empty the heap and set key(T) = ∞ for every old tree T with finite key (these are the trees that have been inserted into the heap during the current growth step).

We can analyze the running time of a pass as follows. The time for the cleanup and other initialization is O(m). If t is the number of old trees, the total time for growing new trees is O(t log k + m): we need at most t delete min operations, each on a heap of size k or smaller, and O(m) other heap operations, none of which is a deletion.

We wish to choose values of k for successive passes so as to minimize the total running time. Smaller values of k reduce the time per pass; larger values reduce the number of passes. For each pass, let us choose $k = 2^{2m/t}$, where m is the original number of edges in the graph and t is the number of trees before the pass. The value of k increases from pass to pass as the number of trees decreases. With this choice of k, the running time per pass is O(m).

It remains for us to bound the number of passes. Consider the effect of a pass that begins with t trees and $m' \leq m$ edges (some edges may have been discarded). Each tree T remaining after the pass has more than $k = 2^{2m/t}$ edges with at least one endpoint in T. (If T_0 is the first old tree among those making up T that was placed in the heap, then T_0 was grown until the heap reached size k, at which time the current tree T' containing T_0 had more than k incident edges. Other trees may have later been connected to T', causing some of these incident edges to have both their endpoints in the final tree T.) Since each of the m' edges has only two endpoints, this means that the number of trees remaining after the pass, say t', satisfies $t' \leq 2m'/k$. If k' is the heap size bound for the next pass, we have $k' = 2^{2m/t'} \geq 2^k$. Since the initial heap size bound is 2m/n and a heap size bound of n or more is only possible on the last pass, the number of passes is at most $\min\{i \mid \log^{(i)} n \leq 2m/n\} + 1 = \beta(m,n) + O(1)$, where $\beta(m,n) = \min\{i \mid \log^{(i)} n \leq m/n\}$ and $\log^{(i)} n$ is defined inductively by $\log^{(0)} n = n, \log^{(i+1)} n = \log \log^{(i)} n$.

Thus we obtain a time bound of $O(m\beta(m,n))$ for the computation of minimum spanning trees. This bound improves the old $O(m \log \log_{(m/n+2)} n)$ bound for all sufficiently sparse graphs. Note that $\beta(m,n) \leq \log^* n$ if $m \geq n$, where $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$. (If $m < n$, then $m = n-1$ and the entire graph is a tree, since we have assumed that the graph is connected.)

Our fast minimum spanning tree algorithm improves the time bounds for certain kinds of constrained minimum spanning tree problems as well. For example, one can find a minimum spanning tree with a degree constraint at one vertex in $O(m\beta(m,n))$ time, and a minimum spanning tree with a fixed number of marked edges in a graph with marked and unmarked edges in $O(n \log n + m)$ time. For details, see Gabow and Tarjan's paper [8].

5. Recent Results and Open Problems

F-heaps have several additional applications. H. Gabow (private communication) has noted that they can be used to speed up the scaling algorithm of Edmonds and Karp [6] for minimum-cost network flow from $O(m^2(\log_{(m/n+2)} n)(\log N))$ to $O(m(n \log n + m)(\log N))$, where N is the maximum capacity, assuming integer capacities. Gabow has also noted that F-heaps speed up the "cheapest insertion" algorithm of Rosenkrantz, Stearns, and Lewis [18] for finding an approximately optimum traveling salesman tour from $O(n^2 \log n)$ to $O(n^2)$, and they can be used to find shortest pairs of

disjoint paths in $O(n \log n + m)$ time, improved from $O(m \log_{(m/n+2)} n)$ [20].

A less immediate application is to the directed analogue of the minimum spanning tree problem, the optimum branching problem. Gabow, et. al.[7] have proposed a very complicated $O(n \log n + m \log\log \log_{(m/n+2)} n)$-time algorithm for this problem, improving on Tarjan's [3,21] bound of $O(\min\{m \log n, n^2\})$. Recently we have found a simple algorithm using F-heaps that runs in $O(n \log n + m)$ time.

Another recent major result is an improvement by Gabow, Galil, and Spencer (private communication) to our minimum spanning tree algorithm of Section 4. They have improved our time bound from $O(m\beta(m,n))$ to $O(m \log\beta(m,n))$ by introducing the idea of grouping edges with a common vertex into "packets" and working only with packet minima.

Several intriguing open questions are raised by our work:

(i) Is there a "self-adjusting version of F-heaps that achieves the same amortized time bounds but requires neither maintaining ranks nor performing cascading cuts? The self-adjusting version of leftist heaps proposed by Sleator and Tarjan [19] does not solve this problem, as the amortized time bound for decrease key is $O(\log n)$ rather than $O(1)$. We have a candidate data structure but are so far unable to verify that it has the desired time bounds.

(ii) Can the best time bounds for finding shortest paths and minimum spanning trees be improved? Dijkstra's algorithm requires $\Omega(n \log n + m)$ time assuming a comparison-based computation model, since it must examine all the edges in the worst case and it can be used to sort n numbers. Nevertheless, this does not preclude the existence of another, faster algorithm. Similarly, there is no reason to suppose that the Gabow-Galil-Spencer bound for minimum spanning trees is best possible. It is suggestive that the minimality of a spanning tree can be checked in $O(m\alpha(m,n))$ time [23], and even in $O(m)$ comparisons [16]. Furthermore if the edges are presorted a minimum spanning tree can be computed in $O(m\alpha(m,n))$ time [24].

(iii) Are there other problems needing heaps where the use of F-heaps gives asymptotically improved algorithms? A possible candidate is the nonbipartite weighted matching problem, for which the current best bound of $O(n^2 \log n + nm \log\log \log_{(m/n+2)} n)$ [7] be improvable to $O(n^2 \log n + nm)$.

Acknowledgement. We thank Dan Sleator for perceptive suggestions that helped to simplify our data structure and Hal Gabow for pointing out the additional applications of F-heaps mentioned in in Section 5.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[2] M. R. Brown, "Implementation and analysis of binomial queue algorithms," SIAM J. Comput. 7(1978), 298-319.

[3] P. M. Camerini, L. Fratta, and F. Maffioli, "A note on finding optimim 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] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numer. Math. 1(1959), 269-271.

[6] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," J. Assoc. Comput. Mach 19(1972), 248-264.

[7] H. N. Gabow, Z. Galil, and T. Spencer, "Efficient implementation of graph algorithms using contraction," Proc. 25th Annual IEEE Symp. on Found. of Comput. (1984).

[8] H. N. Gabow and R. E. Tarjan, "Efficient algorithms for a family of matroid intersection problems," J. Algorithms 5(1984), 80-131.

[9] Z. Galil, S. Micali, and H. Gabow, "Maximal weighted matching on general graphs," Proc. 23rd Annual IEEE Symp. on Found. of Computer Science (1982), 255-261.

[10] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," Ann. History of Computing, to appear.

[11] V. Jarnik, "O jistém problému minimalnim, Práca Moravské Přírodovědecké Společnosti," 6(1930), 57-63. (In Czech.)

[12] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," J. Assoc. Comput. Mach. 24(1977), 1-13.

[13] D. E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Second Edition, Addison-Wesley, Reading, MA 1973.

[14] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

[15] D. E. Knuth, "A generalization of Dijkstra's algorithm," Inform. Process. Lett. 6(1977), 1-5.

[16] J. Komlós, "Linear verification for spanning trees," Proc. 25th Annual IEEE Symp. on Found. of Comput. (1984).

[17] R. C. Prim, "Shortest connection networks and some generalizations," Bell System Tech. J. 36(1957), 1389-1401.

[18] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," SIAM J. Comput. 6(1977), 115-124.

[19] D. D. Sleator and R. E. Tarjan, "Self-adjusting heaps," SIAM J. Comput., submitted.

[20] J. W. Suurballe and R. E. Tarjan, "A quick method for finding shortest pairs of paths," Networks 14(1984), 325-336.

[21] R. E. Tarjan, "Finding optimum branchings," Networks 7(1977), 25-35.

[22] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," J. Comput. System Sci. 18(1979), 110-127.

[23] R. E. Tarjan, "Applications of path compression on balanced trees," J. Assoc. Comput. Mach. 26(1979), 690-715.

[24] R. E. Tarjan, Data Structures and Network Algorithms, Soc. Ind. Appl. Math., Philadelphia, PA, 1983.

[25] R. E. Tarjan, "Amortized computational complexity," SIAM J. Alg. Disc. Meth., to appear.

[26] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," J. Assoc. Comput. Mach. 31(1984), 245-281.

[27] J. Vuillemin, "A data structure for manipulating priority queues," Comm. ACM 21(1978), 309-314.

[28] A. Yao, "An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees," Inform. Process. Lett. 4(1975), 21-23.