

Faster Algorithms for the Shortest Path Problem

RAVINDRA K. AHUJA

Massachusetts Institute of Technology, Cambridge, Massachusetts

KURT MEHLHORN

Universität des Saarlandes, Saarbrücken, West Germany

JAMES B. ORLIN

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

ROBERT E. TARJAN

Princeton University, Princeton, New Jersey and AT&T Bell Laboratories, Murray Hill, New Jersey

Abstract. Efficient implementations of Dijkstra's shortest path algorithm are investigated. A new data structure, called the *radix heap*, is proposed for use in this algorithm. On a network with n vertices, m edges, and nonnegative integer arc costs bounded by C , a one-level form of radix heap gives a time bound for Dijkstra's algorithm of $O(m + n \log C)$. A two-level form of radix heap gives a bound of $O(m + n \log C / \log \log C)$. A combination of a *radix heap* and a previously known data structure called a *Fibonacci heap* gives a bound of $O(m + n\sqrt{\log C})$. The best previously known bounds are $O(m + n \log n)$ using Fibonacci heaps alone and $O(m \log \log C)$ using the priority queue structure of Van Emde Boas et al. [17].

Categories and Subject Descriptors: G.2.2 [Discrete Mathematics]: Graph Theory—*network problems*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Heap, priority queue, shortest paths

1. Introduction

Let $G = (V, E)$ be a graph with vertex set V of size n and arc set E of size m . Let s be a distinguished vertex of G and let c be a function assigning a nonnegative real-valued *cost* to each arc of G . We denote the cost of $(v, w) \in E$ by $c(v, w)$ to avoid extra parentheses. The *single-source shortest path problem* is that of computing,

This research was conducted while R. K. Ahuja was on leave from the Indian Institute of Technology, Kanpur, India. The research of R. K. Ahuja and J. B. Orlin was partially supported by a National Science Foundation (NSF) Presidential Young Investigator Award, contract 8451517 ECS, by Air Force Office of Scientific Research grant AFORS-88-0088, and grants from Analog Devices, Apple Computer, Inc., and Prime Computer. The research of K. Mehlhorn was partially supported by grant DFG Sonderforschungsbereich 124, TB2. The research of R. E. Tarjan was partially supported by NSF grant DCR 86-05962 and Office of Naval Research (ONR) contract N00014-87-K-0467.

Authors' present addresses: R. K. Ahuja and J. B. Orlin, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139; K. Mehlhorn, FB 10, Universität des Saarlandes, 66 Saarbrücken, West Germany; R. E. Tarjan, Department of Computer Science, Princeton University, Princeton, NJ 08544 and AT&T Bell Laboratories, Murray Hill, NJ 07974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 0004-5411/90/0400-0213 \$01.50

for each vertex v reachable from s , the cost of a minimum-cost path from s to v . (The cost of a path is the sum of the costs of its edges.) We assume that all vertices are reachable from s ; if this is not the case, unreachable vertices can be deleted from G in a linear-time preprocessing step.

The theoretically most efficient known algorithm for this problem is Dijkstra's algorithm [6]. Our description of his algorithm is based on that in Tarjan's monograph [13]. The algorithm maintains a *tentative cost* $d(v)$ for each vertex v , such that some path from s to v has total cost $d(v)$. As the algorithm proceeds, the tentative costs decrease, until at the termination of the algorithm, for each vertex v , $d(v)$ is the cost of a minimum-cost path from s to v . Initially $d(s) = 0$ and $d(v) = \infty$ for every $v \neq s$. The algorithm maintains a partition of the vertices into three states: *unlabeled* vertices, those with infinite tentative costs; *labeled* vertices, those with finite tentative cost whose minimum cost is not yet known; and *scanned* vertices, those whose minimum cost is known. Initially, s is labeled and all other vertices are unlabeled. The algorithm consists of repeating the following step until all vertices are scanned:

Scan a Vertex. Select a labeled vertex v such that $d(v)$ is minimum and declare v scanned. For each arc (v, w) , if $d(v) + c(v, w) < d(w)$, replace $d(w)$ by $d(v) + c(v, w)$ and declare w labeled if it is currently unlabeled.

The algorithm can easily be augmented to compute actual minimum-cost paths instead of just the costs of such paths. This computation requires only $O(m)$ additional time.

The key to efficient implementation of Dijkstra's algorithm is the use of a data structure called a *heap* (or *priority queue*). A heap consists of a set of items, each with an associated real-valued *key*, on which the following operations are possible:

- (i) *insert*(h, x). Insert new item x , with predefined key, into heap h .
- (ii) *delete min*(h). Find an item of minimum key in heap h , delete it from h , and return it as the result of the operation.
- (iii) *decrease*($h, x, value$). Replace by *value* the key of item x in heap h ; *value* must be smaller than the old key of x .

In a heap-based implementation of Dijkstra's algorithm, a heap h contains all the labeled vertices; the tentative cost of a labeled vertex is its key. Initially, $h = \{s\}$. The scanning step is implemented as follows:

Scan a Vertex. Let $v = \text{delete min}(h)$. Declare v scanned. For each arc (v, w) , if $d(w) = \infty$, let $d(w) = d(v) + c(v, w)$ and perform *insert*(h, w); if $d(w) < \infty$ and $d(v) + c(v, w) < d(w)$, perform *decrease*($h, w, d(v) + c(v, w)$).

Dijkstra's algorithm runs in $O(m)$ time plus the time required to perform the heap operations. There are n *insert* operations (counting one to insert s initially), n *delete min* operations, and at most $m - n + 1$ *decrease* operations. Dijkstra's original implementation uses an array to represent the heap, giving a bound of $O(1)$ time per *insert* or *decrease* and $O(n)$ time per *delete min*, or $O(n^2)$ time overall. A more modern heap implementation, the Fibonacci heap [8], needs $O(1)$ time per *insert* or *decrease* and only $O(\log n)$ per *delete min*, for an overall time bound of $O(m + n \log n)$. The same bound is attainable using relaxed heaps [7] or Vheaps [12].

A time of $O(m + n \log n)$ is best possible for Dijkstra's algorithm, if the arc costs are real numbers and only binary comparisons are used in the heap implementation. This is because it is easy to reduce the problem of sorting n numbers to a run

of Dijkstra's algorithm. The question arises whether the $O(m + n \log n)$ bound can be beaten in the special case that all the arc costs are integers of moderate size. This is the question we explore in this paper.

Henceforth, we assume that all arc costs are integers bounded above by C . Under this assumption, a data structure of Van Emde Boas et al. [16, 17] can be used to implement the heap in Dijkstra's algorithm, giving a time bound of $O(\log \log C)$ per heap operation, or $O(m \log \log C)$ time in total. The space needed for the heap is $O(n + C)$, but this can be reduced to $O(n + C^\epsilon)$ for any positive constant ϵ using tries [15], or even to $O(n)$ if universal hashing [2] or dynamic perfect hashing [5] is used. (Use of hashing makes the algorithm randomized instead of deterministic and the time bound expected instead of the worst case.)

The existence of an $O(m + n \log n)$ bound for arbitrary real-valued costs suggests the problem of obtaining a bound for integer costs of the form $O(m + nf(C))$ for some function f of the number sizes, with f growing as slowly as possible. An algorithm independently discovered by Dial [4] and Johnson [9] runs in $O(m + nC)$ time. Based on the existence of the Van Emde Boas–Kaas–Zijlstra data structure, one might hope for a bound of $O(m + n \log \log C)$. Obtaining such a bound is an open problem. We shall develop a data structure that results in a bound of $O(m + n\sqrt{\log C})$. Our data structure, the *radix heap*, exploits special properties of the heap operations in Dijkstra's algorithm. The most important of these properties is that successive *delete min* operations return vertices in nondecreasing order by tentative cost. The simplest form of the data structure, the *one-level radix heap*, was originally proposed by Johnson [10], who used it to obtain an $O(m \log \log C + n \log C \log \log C)$ time bound for Dijkstra's algorithm. By slightly changing the implementation, we reduce the time to $O(m + n \log C)$. Section 2 describes this result.

By adding another level to the structure, we obtain a *two-level radix heap*. The idea of adding a second level is borrowed from Denardo and Fox [3]. The new structure reduces the running time of Dijkstra's algorithm to $O(m + n \log C / \log \log C)$. Section 3 presents this result. One more change to the structure, the addition of Fibonacci heaps in the second level, reduces the time bound further, to $O(m + n\sqrt{\log C})$. Section 4 discusses this improvement.

Section 5 discusses the effect of increasing the cost of doing arithmetic; all the results mentioned above are predicated on the assumption that integers of size $O(nC)$ can be added or compared in constant time. In the semilogarithmic model studied in Section 5, in which arithmetic on integers of $O(\log n)$ bits takes $O(1)$ time, the m -term in the bounds given above increases to $O(m \log C / \log n)$, while the n -term remains the same.

2. One-Level Radix Heaps

Radix heaps rely on the following properties of Dijkstra's algorithm:

- (i) For any vertex v , if $d(v)$ is finite, $d(v) \in [0 \dots nC]$.¹
- (ii) For any vertex $v \neq s$, if v is labeled, $d(v) \in [d(x) \dots d(x) + C]$, where x is the most recently scanned vertex.

Property (ii) implies in particular that successive *delete min* operations return vertices in nondecreasing order by tentative cost.

A *one-level radix heap* is a collection of $B = \lceil \lg(C + 1) \rceil + 2$ buckets,² indexed from 1 through B . Each bucket has an associated *size*. The size of bucket i is

¹ We denote the interval of integers $\{x \mid l \leq x \leq u\}$ by $[l \dots u]$.

² We denote \log_2 by \lg .

denoted by $size(i)$ and defined as follows:

$$\begin{aligned} size(1) &= 1; \\ size(i) &= 2^{i-2} \quad \text{for } 2 \leq i \leq B - 1; \\ size(B) &= nC + 1. \end{aligned}$$

Observe that the bucket sizes satisfy the following important inequality:

$$\sum_{i=1}^{j-1} size(i) \geq \min\{size(j), C + 1\} \quad \text{for } 2 \leq j \leq B. \quad (1)$$

Each bucket also has a *range* that is an interval of integers. Initially the ranges of the buckets partition the interval $[0 \dots nC + 1]$. In general the ranges partition the interval $[d_{\min} \dots nC + 1]$, where d_{\min} is the maximum label of a scanned node. For each bucket i the upper bound $u(i)$ of its interval is maintained; the range of bucket i is $range(i) = [u(i - 1) + 1 \dots u(i)]$, with the conventions that $u(0) = d_{\min} - 1$ and $range(i) = \emptyset$ if $u(i - 1) \geq u(i)$. Whereas the sizes of all buckets are fixed throughout the computation, the ranges change; for each bucket i , $u(i)$ is a nondecreasing function of time.

Initially $u(i) = 2^{i-1} - 1$ for $1 \leq i \leq B - 1$, $u(B) = nC + 1$. Observe that this implies $|range(i)| \leq size(i)$. This inequality is maintained throughout the computation for each bucket. The labeled vertices are stored in the buckets, with vertex v stored in bucket i if $d(v) \in range(i)$. Initially, vertex s is inserted into bucket 1. The range of bucket 1 is maintained so that every vertex v in bucket 1 has $d(v) = u(1)$; thus the effective range of bucket 1 contains only $u(1)$.

Each bucket is represented by a doubly linked list of its vertices, to make insertion and deletion possible in constant time. In addition, stored with each vertex is the index of the bucket containing it.

The three heap operations are implemented as follows. To insert a newly labeled vertex v , examine values of i in decreasing order, starting with $i = B$, until finding the largest i such that $u(i) < d(v)$; then insert v into bucket $i + 1$. To decrease the key of a vertex v , remove v from its current bucket, say bucket j . Reduce the key of v . Proceed as in an insertion to reinsert v into the correct bucket, but begin with bucket $i = j$.

For a single vertex v , the time spent on insertion and all *decrease* operations is $O(\log C)$ plus $O(1)$ per *decrease*, because the index of the bucket containing v can never increase. Thus, the total time for all such operations during a run of Dijkstra's algorithm is $O(m + n \log C)$.

The most complicated operation is *delete min*, which is performed as follows. If bucket 1 is nonempty, return any vertex in bucket 1. Otherwise, find the nonempty bucket of smallest index, say bucket j . By scanning through the items in bucket j , find a vertex of smallest tentative cost, say v . Save v to return as the result of the *delete min* and distribute the remaining vertices in bucket j among buckets of smaller index, as follows. Replace $u(0)$ by $d(v) - 1$, $u(1)$ by $d(v)$, and for i running from 2 through $j - 1$, replace $u(i)$ by $\min\{u(i - 1) + size(i), u(j)\}$. Remove each vertex from bucket j and reinsert it as in *decrease*; do not reinsert v .

Property (ii) and inequality (1) guarantee that, if $j \geq 2$ in a *delete min*, every vertex in bucket j will move to a bucket of strictly smaller index. It follows that the time spent on *delete min* operations is $O(\log C)$ per *delete min* plus $O(\log C)$ per vertex, for a total of $O(n \log C)$ during a run of Dijkstra's algorithm. We conclude that the total running time of Dijkstra's algorithm with this implementation is

$O(m + n \log C)$. The space required is $O(m + \log C)$. Johnson [10], using the same data structure, obtained a bound worse by a factor of $\log \log C$ because he used binary search instead of sequential scan to reinsert vertices into buckets.

3. Two-Level Radix Heaps

Reducing the running time of the algorithm of Section 2 requires reducing the number of reinsertions of vertices into buckets. This can be done by increasing the bucket sizes, but then inequality (1) no longer holds. We overcome this problem by dividing each bucket into *segments*. All segments within a bucket have the same size.

A *two-level radix heap* is defined by a parameter K , determining the number of segments within a bucket. The number of buckets is $B = \lceil \log_K(C + 1) \rceil + 1$. The sizes of the buckets are as follows:

$$\begin{aligned} \text{size}(i) &= K^i & \text{for } 1 \leq i \leq B - 1; \\ \text{size}(B) &= nC + 1. \end{aligned}$$

As in the one-level scheme, bucket i has $\text{range}(i) = [u(i - 1) + 1 \dots u(i)]$, with $u(0) = d_{\min} - 1$ and $u(B) = nC + 1$. The remaining upper bounds on ranges have the following initial values:

$$u(j) = \sum_{i=1}^j K^i - 1 \quad \text{for } 1 \leq j \leq B - 1.$$

For $1 \leq i \leq B - 1$, bucket i is partitioned into K segments, each of size K^{i-1} . Segments are indexed by ordered pairs; segment (i, k) is segment k of bucket i . Bucket B consists of a single segment.

Each segment has an associated range, which is a function of the range of its bucket. The range of segment (i, k) is $\text{range}(i, k) = [u(i, k - 1) + 1 \dots u(i, k)]$ where $u(i, k)$ is defined as follows:

$$u(i, k) = \max\{u(i - 1), u(i) - (K - k)K^{i-1}\}.$$

Observe that $|\text{range}(i, k)| \leq K^{i-1} = \text{size}(i, k)$ for $1 \leq i \leq B - 1$ and $1 \leq k \leq K$. The algorithm maintains the invariant that for $1 \leq i \leq B - 1$, $|\text{range}(i)| \leq K^i$.

The algorithm maintains the ranges of buckets (i.e., the $u(i)$'s) explicitly, but computes the ranges of segments as needed. Observe that, given an integer $x \in \text{range}(i)$, the value of k such that $x \in \text{range}(i, k)$ can be computed in constant time using the formula

$$k = K - \left\lfloor \frac{u(i) - x}{K^{i-1}} \right\rfloor.$$

Choosing K to be a power of two simplifies this computation on a computer whose number representation is binary, but this is not necessary for our theoretical results.

The labeled vertices are stored in the segments, with vertex v stored in segment (i, k) if $d(v) \in \text{range}(i, k)$. Each segment is represented by a doubly linked list. The three heap operations are implemented as follows. An *insert* or *decrease* operation on a vertex v is performed as in a one-level heap, except that once the bucket i such that $d(v) \in \text{range}(i)$ is located, the k such that $d(v) \in \text{range}(i, k)$ is computed (in constant time), and v is inserted into segment (i, k) . The total

time for all *insert* and *decrease* operations during a run of Dijkstra's algorithm is $O(m + Bn) = O(m + n \log_K C)$.

The *delete min* operation is implemented much as in a one-level heap, except that only the contents of a single segment are distributed, not the contents of an entire bucket. To perform *delete min*, find the first nonempty bucket, say j . Find the first nonempty segment within bucket j , say (j, k) . (If $j = B$, $k = 1$, since bucket B consists of only a single segment.) If $j = 1$, remove and return any vertex in segment (j, k) . Otherwise, scan the vertices in segment (j, k) to find one, say v , with minimum tentative cost. Redefine $u(i)$ for $0 \leq i \leq j - 1$ as in a one-level heap. Distribute all vertices in segment (j, k) (except v) into their new correct segments, which lie in buckets 1 through $j - 1$.

A few details of the data structure deserve comment. To facilitate locating the first nonempty bucket, a bit for each bucket is maintained that indicates whether or not the bucket is empty. Determining j in a *delete min* then takes $O(B)$ time. The segments are represented as an array of doubly linked lists, with the index of segment (i, k) being $K(i - 1) + k$. Since each vertex in a segment that is distributed moves to a lower bucket, the total number of such movements is $O(Bn)$. The total time for all the *delete min* operations is $O(Bn)$ plus the time for n steps of the form, "find the first nonempty segment in a given bucket."

If each such segment is found merely by scanning all the segments in the bucket, the time for one such step is $O(K)$, and the total running time of Dijkstra's algorithm is $O(m + (B + K)n)$. Choosing K proportional to $\log C / \log \log C$ gives $B = \lceil \log_K(C + 1) \rceil + 1 = O(\log C / \log \log C)$, and the total running time is $O(m + n \log C / \log \log C)$. The space required is $O(m + (\log C / \log \log C)^2)$, reducible to $O(m + (\log C / \log \log C)^\epsilon)$ for any constant $\epsilon > 0$ using a trie [15] or even to $O(m)$ using either universal hashing [2] or dynamic perfect hashing [5].

If $C < n$, the running time of the algorithm can be reduced to $O(m + n \log C / \log \log n)$ by using table lookup to find nonempty segments. Specifically, choose $K = \lceil \lg n \rceil$. For each bucket (other than bucket B), maintain an integer of $\lceil \lg n \rceil$ bits whose k th bit is one if segment k of the bucket is nonempty, and zero otherwise. During a preprocessing step, construct an array of n positions, indexed from 1 to n , such that position i contains k if and only if the k th bit of i (expressed in binary) is the first nonzero bit. Construction of this array takes $O(n)$ time, and once the array is constructed, the first nonempty segment of a nonempty bucket can be found in $O(1)$ time by accessing the array position indexed by the integer encoding the nonempty segments.

By choosing the appropriate one of the two methods above for finding the first nonempty segment in a bucket, we obtain a time bound of $O(m + n \log C / \log \log (nC))$ for Dijkstra's algorithm.

4. Use of Fibonacci Heaps

Our final improvement reduces the running time of Dijkstra's algorithm to $O(m + n\sqrt{\log C})$ by using a variant of Fibonacci heaps to find nonempty segments. Throughout this section we shall refer to each segment by its index; as in Section 3, the index of segment (i, k) is $K(i - 1) + k$, which is an integer in the interval $[1 \dots KB - K + 1]$. We associate with each labeled vertex the index of the segment containing it. We need to be able to maintain the collection of labeled vertices under the following three kinds of operations:

- (i) *delete min*. Find a labeled vertex of minimum index, mark it scanned, and return it.

- (ii) *insert*(x). Declare x to be a newly labeled vertex, with predefined index.
- (iii) *decrease*(x , *value*). Replace the index of labeled vertex x by *value*; *value* must be smaller than the old index of x .

In other words, we must maintain the set of labeled vertices as a heap, with the key of each vertex equal to its index. A run of Dijkstra's algorithm requires n *insert* operations, n *delete min* operations, and at most m *decrease* operations, in addition to the time for maintaining bucket boundaries and recomputing segments. The total time for all the latter bookkeeping is $O(m + Bn)$.

Fibonacci heaps (abbreviated F-heaps) support *delete min* in $O(\log n)$ amortized time³ and *insert* and *decrease* in $O(1)$ amortized time [8], where n is the maximum heap size. But in our application, the number of possible index values is much smaller than the number of vertices. We shall describe how to extend Fibonacci heaps so that if the keys are integers in the interval $[1 \dots N]$, the amortized time per *delete min* is $O(\log \min\{n, N\})$, while the amortized time per *insert* or *decrease* remains $O(1)$. In the application at hand, we can take $N = KB$. The choice of $K = 2^{\lceil \sqrt{\log C} \rceil}$ gives $B = O(\log_K C) = O(\sqrt{\log C})$, and $\log N = O(\sqrt{\log C})$; therefore, the total running time of Dijkstra's algorithm is $O(m + n\sqrt{\log C})$.

It remains for us to make the necessary changes to F-heaps. The main idea is to make sure that such a heap contains at most N items, that is, at most one item per key value. Making this idea work in the presence of *decrease* operations requires some care and some knowledge of the internal workings of F-heaps.

We need to know the following facts about F-heaps. An F-heap consists of a collection of heap-ordered trees whose nodes are the items in the heap. (A heap-ordered tree is a rooted tree such that if $p(x)$ is the parent of node x , the key of x is no less than the key of $p(x)$.) Each node in an F-heap has a *rank* equal to the number of its children. A fundamental operation on F-heaps is *linking*, which combines two heap-ordered trees into one by comparing the keys of their roots and making the root of smaller key the parent of the root of larger key, breaking a tie arbitrarily. A link operation takes $O(1)$ time. Only trees with roots of equal rank are linked.

Each nonroot node in an F-heap is in one of two states, *marked* or *unmarked*. When a node becomes a nonroot by losing a comparison during a link, it becomes unmarked. Nodes become marked during *decrease* operations, as described below.

The three heap operations are performed as follows. To insert a new item, merely make it into a one-node tree and add this tree to the collection of trees. This takes $O(1)$ time.

To perform a *decrease* operation on a node x , begin by updating the key of x . Then, if x is not a root, cut the edge joining x and $p(x)$ and repeat the following step, with y initially equal to the old $p(x)$, until y is unmarked or y is a tree root: cut the edge joining y and its parent $p(y)$, and set y equal to $p(y)$. After the last such cut, if the last node y is not a root, mark it.

The overall effect of such a *decrease* operation is possibly to break the initial tree containing x into several trees, one of which has root x . The time required by the *decrease* operation is $O(1)$ plus $O(1)$ per cut. Since only one node is marked per decrease operation, and since one node becomes unmarked per cut except for at most one cut per *decrease* operation, the total number of cuts during a sequence of *decrease* operations is at most twice the number of *decrease* operations, even though a single *decrease* can result in many cuts.

³ By *amortized time*, we mean the time per operation averaged over a worst-case sequence of operations. See Tarjan's survey paper [14] or Mehlhorn's book [11].

To perform the third heap operation, *delete min*, scan all the tree roots and identify one, say x , of minimum key. Remove x from its tree, thereby making each of its children a tree root. Finally, repeatedly link trees whose roots have equal rank, until no two tree roots have equal rank.

The key to the analysis of F-heaps is that manipulation of rooted trees in the ways described above maintains the following invariant: for any node x , $rank(x) = O(\log size(x))$, where $size(x)$ is the number of nodes in the subtree rooted at x . A simple analysis gives an amortized time bound of $O(1)$ for *insert* and *decrease*, and $O(\log n)$ for *delete min*.

Now we extend F-heaps to reduce the amortized time per *delete min* to $O(\log \min\{n, N\})$. For each value $i \in [1..N]$, the algorithm maintains the set $S(i)$ of items with key i . One item in $S(i)$ is designated the *representative* of $S(i)$. All the items, both the representatives and the nonrepresentatives, are grouped into heap-ordered trees of the kind manipulated by the F-heap algorithm. These trees are divided into two groups: *active trees*, those whose roots are representatives, and *passive trees*, those whose roots are nonrepresentatives.

The algorithm maintains the following two invariants:

- (i) The key of a nonroot node x is strictly greater than that of its parent (a strengthening of the heap order property);
- (ii) Every nonrepresentative is a root.

Invariant (ii) implies that all nodes in active trees are representatives and hence have distinct keys; thus, the number of nodes in active trees is at most N . Invariants (i) and (ii) together imply that the representative of minimum key is the root of an active tree; hence, *delete min* need only scan the roots of active trees.

The three heap operations are performed as follows: To insert an item x , make it into a one-node tree, which becomes active or passive depending on whether the set $S(i)$ into which x is inserted is empty or not; if it is, x becomes the representative of $S(i)$. To perform a *decrease* operation, proceed as on an ordinary F-heap as described above, with the following addition: move x from its old set, say $S(i)$, to the appropriate new set, say $S(j)$. Make some other item (if any) in $S(i)$, say y , the representative of $S(i)$ and make the tree with root y active. If x is the only item in $S(j)$, make the tree rooted at x active; otherwise, make it passive. Make other new trees created by cuts active. The total time required by a *decrease* operation is $O(1)$ plus $O(1)$ per cut, including the time to move trees between the active and passive groups.

To perform *delete min*, proceed as on an ordinary F-heap, with the following changes: Scan only the roots of active trees to find a minimum, say x . Delete x from the set $S(i)$ containing it, and if $S(i)$ remains nonempty, choose some item in $S(i)$, say y , to be the new representative; activate the tree with root y . Then perform repeated linking, but only on active trees; that is, after deleting the active node of minimum key and updating the representative of its set, repeatedly link active trees whose roots have equal rank until all active trees have roots of different ranks.

The efficiency analysis of the extended data structure is almost the same as that of the original. Define the *potential* of the data structure to be the number of trees plus twice the number of marked nodes. Define the *amortized time* of a heap operation to be its actual time (measured in suitable units) plus the net increase in potential it causes. The initial potential is zero (if the initial heap is empty) and the potential is always nonnegative. It follows that, for any sequence of heap operations, the total amortized time is an upper bound on the total actual time.

The amortized time of an insertion is $O(1)$, since it increases the potential by one. A *decrease* operation causing k cuts adds $O(1) - k$ to the potential: each cut except for at most one adds a tree but removes a marked node; marked nodes count for two in the potential. Thus, a decrease takes $O(1)$ amortized time if a cut is regarded as taking unit time.

Each link during a *delete min* operation reduces the potential by one and thus has an amortized time of zero, if a link is regarded as taking unit time. Not counting links, the time spent during a *delete min* is $O(\log \min\{n, N\})$, as is the increase in potential caused by removing a node of minimum key: the maximum rank of any node is $O(\log \min\{n, N\})$ by the same argument used in the analysis of ordinary F-heaps. Thus, the amortized time of *delete min* is $O(\log \min\{n, N\})$, as desired.

The idea used here, that of grouping trees into active and passive, applies as well to Vheaps [12] to give the same time bounds, but it does not seem to apply to relaxed heaps [7]. The extended F-heap, if used directly in the implementation of Dijkstra's algorithm, gives a running time of $O(m + n \log C)$, the same as that obtained in Section 2.

5. Time Bounds in a Semilogarithmic Computation Model

In the previous sections, we analyzed our algorithms using a unit-cost random-access machine [1] as the computation model. In particular, we assumed that addition and comparison of integers in the interval $[0 \dots nC]$ takes $O(1)$ time. If C is large, this assumption may not be realistic. In this section, we derive bounds for the algorithms assuming a semilogarithmic-cost computation model. We show that in this model, the m -term in our bounds becomes $m \lceil \lg C / \lg n \rceil$, while the n -term remains unchanged. The following two assumptions define the semilogarithmic model:

- (1) Arithmetic on integers of length $O(\log n)$ and all other random access machine operations (index calculations, pointer assignments, etc.) take $O(1)$ time;
- (2) $\log C = n^{\alpha(1)}$.

In our algorithms, we represent arc costs and tentative costs $d(v)$ as arrays of length $\lceil \lg(nC + 1) / R \rceil$, where $R = \lfloor \lg n / 2 \rfloor$. Each array element is an integer in the range $[0 \dots 2^R - 1]$. By assumption (1), indexing into these arrays takes $O(1)$ time, but this is only reasonable if the indices are $O(\log n)$ in length, which is the reason for imposing assumption (2). Henceforth, in this section we also assume that $\log C = \Omega(\log n)$; if $\log C = O(\log n)$, the bounds of the previous sections hold without change for the semilogarithmic model. If $\log C = \Omega(\log n)$, then $\lg(nC + 1) = O(\log C)$, a fact that we shall use repeatedly without further comment.

Let us first analyze the algorithm of Section 2. We shall revise and reformulate the algorithm to fit into the semilogarithmic model better. In particular, we emphasize the bit manipulation involved in the computation. Let $B = \lceil \lg(nC + 1) \rceil$. At a given time in the algorithm, let V be a labeled vertex with minimum tentative cost $d(v)$. Let $\alpha_{B-1} \dots \alpha_1$ be the binary representation of $d(v)$; that is, $\alpha_i \in \{0, 1\}$ and $d(v) = \sum_{i=1}^{B-1} \alpha_i 2^{i-1}$. The algorithm maintains buckets numbered 1 through B containing the labeled vertices, with bucket 1 containing every vertex u such that $d(u) = d(v)$ and bucket i , for $2 \leq i \leq B$, containing every vertex u such that position $i - 1$ is the largest position in which the binary representations of $d(u)$ and $d(v)$ differ. Note that in this variant of the Section 2 algorithm, bucket ranges are represented implicitly rather than explicitly.

Finding the smallest nonempty bucket by a sequential scan over the buckets takes $O(\log(nC + 1)) = O(\log C)$ time. Distributing the vertices in a bucket is done by scanning down through the appropriate bits of the tentative costs of the vertices in the bucket. Such distribution takes $O(\log(nC + 1)) = O(\log C)$ time per vertex over the entire algorithm. (Extracting the appropriate bit of a tentative cost can be done either by appropriate shifting and masking operations, or, if these are not available, by table lookup. In either case, the time to extract a bit is $O(1)$.) Updating tentative costs takes $O(m \log C / \log n)$ time over the entire algorithm. It follows that the total running time of Dijkstra's algorithm is $O(m \log C / \log n + n \log C)$.

Next we turn to the algorithms of Sections 3 and 4. In these algorithms each bucket is divided into K segments. In the spirit of assumption (1), we restrict ourselves to $\log K = O(\log n)$. The number of buckets is $B = \lceil \log_K(nC + 1) \rceil$. The assignment of labeled vertices to buckets and segments is as follows. Let $\alpha_{B-1} \cdots \alpha_0$ be the K -ary expansion of the minimum tentative cost of a labeled vertex, say v . A labeled vertex u belongs to segment k of bucket i if either $i = 1$, $k = \alpha_0 + 1$, and $d(u) = d(v)$; or if $i - 1$ is the largest position at which the K -ary expansions of $d(u)$ and $d(v)$ differ and $k = \alpha_{i-1} + 1$.

The time to find the first nonempty segment (by scanning over buckets, then over segments within a bucket) is $O(B + K)$. The total time for distributing vertices among segments is $O(B)$ per vertex. The total running time of the method is thus $O(m \log C / \log n + nB + nK)$. Choosing K proportional to $\lg(nC) / \lg \lg(nC)$ gives a total running time of $O(m \log C / \log n + n \log(nC) / \log \log(nC)) = O(m \log C / \log n + n \log C / \log \log C)$.

Adding an extended F-heap to represent the nonempty segments, as in Section 4, reduces the time to find the first nonempty segment to $O((\log BK)^2 / \log n)$: there are $O(\log BK)$ steps, each of which manipulates integers in the interval $[1 \dots BK]$, which is the range of the segment indices.) The m decrease operations on the F-heap take $O(m \log(BK) / \log n)$ time. The total time to run Dijkstra's algorithm is thus

$$O\left(m\left(\frac{\log C}{\log n} + \frac{\log(BK)}{\log n}\right) + n\left(B + \frac{(\log(BK))^2}{\log n}\right)\right).$$

Choosing $K = 2^{\lceil \sqrt{\lg(nC+1)} \rceil}$ if $\lg(nC + 1) \leq (\lg n)^2$, $K = n$ if $\lg(nC + 1) > (\lg n)^2$ gives a total running time of $O(m \log C / \log n + n\sqrt{\log C})$.

It is worthwhile to compare these bounds with the running time of the straight F-heap-based algorithm [8]. That algorithm requires $O(m + n \log n)$ steps, each of which involves addition or comparison of integers in the range $[0 \dots nC]$ and thus takes $O(\log C / \log n)$ time in the semilogarithmic model. The total time of the algorithm is thus $O(m \log C / \log n + n \log C)$, the same as the time for the modified Section 2 algorithm. The F-heap algorithm is more complicated, however. The algorithms of Sections 3 and 4 are both faster than the F-heap algorithm, for appropriate values of the parameters. Note that the time to read the problem input is $\Omega(m \log C / \log n)$. Assuming that solving the problem requires reading the input, the algorithm that combines a two-level distributive heap with an F-heap is optimum to within a constant factor if

$$\log C = \Omega\left(\left(\frac{n \log n}{m}\right)^2\right).$$

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

2. CARTER, J. L., AND WEGMAN, M. N. Universal classes of hash functions. *J. Comput. Syst. Sci.* 18 (1979), 143–154.
3. DENARDO, E. V., AND FOX, B. L. Shortest-route methods: 1. Reaching, pruning, and buckets. *Oper. Res.* 27 (1979), 161–186.
4. DIAL, R. Algorithm 360: Shortest path forest with topological ordering. *Commun. ACM* 12 (1969), 632–633.
5. DIETZFELBINGER, M., KARLIN, A., MEHLHORN, K., MEYER AUF DER HEIDE, F., ROHNERT, H., AND TARJAN, R. Dynamic perfect hashing: Upper and lower bounds. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1988, pp. 524–531.
6. DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.
7. DRISCOLL, J. R., GABOW, H. N., SHRAIRMAN, R., AND TARJAN, R. E. Relaxed heaps: An alternative to Fibonacci heaps. *Commun. ACM* 31 (1988), 1343–1354.
8. FREDMAN, M. L., AND TARJAN, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34 (1987), 596–615.
9. JOHNSON, D. B. Efficient algorithms for shortest paths in sparse networks. *J. ACM* 24 (1977), 1–13.
10. JOHNSON, D. B. Efficient special-purpose priority queues. In *Proceedings of the 15th Annual Allerton Conference on Communications, Control, and Computing* (1977), pp. 1–7.
11. MEHLHORN, K. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, New York, N.Y., 1984.
12. PETERSON, G. L. A balanced tree scheme for meldable heaps with updates. Tech. Rep. GIT-TCS-87-23. School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Ga., 1987.
13. TARJAN, R. E. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1983.
14. TARJAN, R. E. Amortized computational complexity. *SIAM J. Alg. Discrete Meth.* 6 (1985), 306–318.
15. TARJAN, R. E., AND YAO, A. C.-C. Storing a sparse table. *Commun. ACM* 22 (1979), 606–611.
16. VAN EMDE BOAS, P. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Proc. Lett.* 6 (1977), 80–82.
17. VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. Design and implementation of an efficient priority queue. *Math. Syst. Theory* 10 (1977), 99–127.

RECEIVED JUNE 1988; REVISED MAY 1989; ACCEPTED JUNE 1989