WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

FASTER ALGORITHMS FOR THE SHORTEST PATH PROBLEM

Ravindra K. Ahuja
Kurt Mehlhorn
James B. Orlin
Robert E. Tarjan

Sloan W.P. No. 2043-88 April 1988

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

FASTER ALGORITHMS FOR THE SHORTEST PATH PROBLEM

Ravindra K. Ahuja
Kurt Mehlhorn
James B. Orlin
Robert E. Tarjan

Sloan W.P. No. 2043-88 April 1988

Faster Algorithms for the Shortest Path Problem

Ravindra K. Ahuja*
Sloan School of Management
M.I.T., Cambridge, MA. 02139 , USA

Kurt Mehlhorn
FB 10, Universität des Saarlandes
66 Saarbrücken,
FEDERAL REPUBLIC OF GERMANY

James B. Orlin
Sloan School of Management
M.I.T., Cambridge, MA. 02139 , USA

Robert E. Tarjan
Department of Computer Science
Princeton University, Princeton, NJ 08544, USA
and
A.T. & T. Bell Laboratories
Murray Hill, NJ 07974 , USA

---

* On leave from Indian Institute of Technology, Kanpur - 208 016 , INDIA

Faster Algorithms for the Shortest Path Problem

Abstract

In this paper, we present the fastest known algorithms for the shortest path problem with nonnegative integer arc lengths. We consider networks with n nodes and m arcs and in which C represents the largest arc length in the network. Our algorithms are obtained by implementing Dijkstra's algorithm using a new data structure which we call a *redistributive heap*. The one-level redistributive heap consists of O(log C) buckets, each with an associated range of integer numbers. Each bucket stores nodes whose temporary distance labels lie in its range. Further, the ranges are dynamically changed during the execution, which leads to a redistribution of nodes to buckets. The resulting algorithm runs in O(m + n log C) time. Using a two-level redistributive heap, we improve the complexity of this algorithm to O(m + n log C/ log log nC). Finally, we use a modified version of Fibonacci heaps to reduce the complexity of our algorithm to $O(m + n\sqrt{\log C})$. This algorithm, under the assumption that the largest arc length is bounded by a polynomial function of n, runs in $O(m + n\sqrt{\log n})$ time, which improves over the best previous strongly polynomial bound of O(m + n log n) due to Fredman and Tarjan. We also analyse our algorithms in the *semi-logarithmic model of computation*. In this model, it takes $\lceil \log x/\log n \rceil$ time to perform arithmetic on integers of value x. It is shown that in this model of computation, some of our algorithms run in linear time for sufficiently large values of C.

The shortest path problem is one of the most well-studied combinatorial optimization problems. Algorithmic developments concerning this problem have proceeded along two directions: (i) development of algorithms that are superior from worst-case complexity point of view (e.g., Dijkstra [1959], Johnson [1977a, 1977b, 1982], Boas, Kaas and Zijlstra [1977], Fredman and Tarjan [1984] , and Gabow [1985]); and (ii) development of algorithms that are very efficient in practice (e.g., Dial [1969], Gilsinn and Witzgall [1973], Dial et. al [1979] , Denardo and Fox [1979], Pape [1980], and Glover et. al [1985]).

Surprisingly, these two directions have not been very complementary. The algorithms that achieve the best worst-case complexity have generally not been attractive empirically , and the algorithms that have performed well in practice have generally failed to have an attractive worst-case bound. In this paper, we present new implementations of Dijkstra's algorithm intended to bridge this gap. Under the assumption that arc lengths are bounded by a polynomial function of n , these algorithms achieve the best possible worst-case complexity for all but very sparse graphs and yet are simple enough to be efficient in practice.

Let $G = (N, A)$ be a directed network with a nonnegative integer arc length c_{ij} associated with each arc $(i, j) \in A$. Let $A(i) = \{(i, j) \in A: j \in N\}$, for each $i \in N$. Let $n = |N|$, $m = |A|$ and $C = \max \{c_{ij} : (i, j) \in A\}$. Further, let s be a distinguished node of the network called the *source*. The shortest path problem is to identify a path of the shortest length from the source s to every other node in the network.

We assume that all logarithms in this paper are base 2, unless stated otherwise. We further assume, except in Section 5, that all arithmetic operations (including multiplication and division) take $O(1)$ time. In all the divisions (multiplications) in the paper, however, the divisor (multiplier) is a power of 2.

Dijkstra's algorithm [1959] is possibly the most well-known method to solve the shortest path problem. Dijkstra's algorithm maintains a distance label $d(j)$ for each node j and a partition of the set of nodes into two subsets: *permanently* labeled nodes and *temporarily* labeled nodes. At each iteration, the algorithm selects a temporarily labeled node i with smallest distance label, makes its label permanent, and scans arcs in $A(i)$ to revise the distance labels of adjacent temporarily labeled nodes. The method stops when all nodes are permanently labeled.

Dijkstra's original implementation of the algorithm runs in $O(n^2)$ time. This bound is the best possible for fully dense networks, but can be improved using clever priority queue data structures if the number of arcs is much less than n^2. The following table indicates such improvements. In the table, $d = \max(2, \lceil m/n \rceil)$ and represents the average degree of a node.

#	Due to	Running Time
1.	Williams [1964]	$O(m \log n)$
2.	Johnson [1977a]	$O(m \log_d n)$,
3.	Johnson [1977b]	$O(m \log\log C + n \log C \log\log C)$
4.	Boas, Kaas, and Zijlstra [1977]	$O(C + m \log\log C)$
5.	Johnson [1982]	$O(m \log\log C)$
6.	Fredman and Tarjan [1984]	$O(m + n \log n)$
7.	Gabow [1985]	$O(m \log_d C)$

Table 1. Running times of polynomial shortest path problems.

For the sake of comparing the above algorithms, we make the reasonable assumption that C is bounded by a polynomial function in n, (i,e., $C = O(n^{O(1)})$. This assumption is known as the *similarity assumption* (Gabow [1985]). Under this assumption, Fredman and Tarjan's implementation using *Fibonacci heaps* runs faster than others for all classes of graphs except very sparse ones, for which the method of Johnson [1982] appears more attractive.

In this paper, we describe a new data structure that we call the *redistributive heap*, and we use it to implement Dijkstra's algorithm. We consider both one-level and two-level versions of this data structure. A one-level redistributive heap consists of O(log C) buckets, each with an associated range of integer numbers. Each bucket stores nodes whose temporary distance labels lie in its range. The ranges of buckets are changed dynamically, which leads to redistribution of nodes among buckets. The resulting algorithm runs in $O(m + n \log C)$ time. Using a two-level (or two-echelon) bucket system, we improve the complexity of our algorithm to $O(m + n \log C / \log\log nC)$. The use of a modified version of the Fibonacci heap data structure further reduces the complexity of our algorithm to $O(m + n\sqrt{\log C})$. Under the similarity assumption, this algorithm runs in $O(m + n\sqrt{\log n})$ time and improves the running time of Fredman and Tarjan's

shortest path algorithm. Further, the first two of our algorithms use very simple data structures which make them attractive from an implementation viewpoint.

Our one-level bucket method uses essentially the same data structure as proposed by Johnson [1977b], except that our search technique is simultaneously simpler and more efficient. Whereas Johnson uses binary search to insert nodes into buckets, we use sequential search. Consequently, his algorithm is O(log log C) times slower than our algorithm. Our two-level bucket approach may be viewed as a hybrid between Johnson's [1982] and Denardo and Fox's [1979] algorithm.

Our algorithms are computationally attractive even if C is not $O(n^{0(1)})$, i.e., the similarity assumption does not hold. In this case, the uniform model of computation, which assumes that all arithmetic operations take time O(1), is arguably inappropriate. It is more realistic to adopt the *semi-logarithmic model of computation* in which arithmetic operations take time proportional to the length of the integers they manipulate. This model of computation assumes that arithmetic on integers of length O(log n) has cost O(1). We also assume that $C = O(2^{n^{0(1)}})$; i. e., $\log C = O(n^{0(1)})$.

We analyse the worst-case complexity of our algorithms in the semi-logarithmic model of computation. In particular, it is shown that our algorithm which uses Fibonacci heaps runs in time $O(m\lceil \log nC/\log n\rceil + n\sqrt{\log nC})$ time. Consequently, this algorithm runs in linear time (and hence is optimal) whenever $\log nC \geq (\log n)^2$ or $m \geq n \log n/\sqrt{\log nC}$. In particular, the algorithm runs in time linear in the size of the data for all sufficiently large C.

1. One-Level Redistributive Heap and Dijkstra's Algorithm

In this section, we describe the Redistributive heap (abbreviated as R-heap) and we use it to implement Dijkstra's algorithm. We illustrate heap operations with the help of a numerical example. Here we describe the one-level version of the R-heap, leaving the generalizations for the subsequent sections.

1.1. Properties of R-heap

A *heap* (or a *priority queue*) is an abstract data structure consisting of items each with a real valued *label*, and on which the following operations can be

performed: create, insert, delete, and find-min. The R-heap is a data type that differs from a heap in a few ways that are particular to the operations needed for Dijkstra's algorithm. The R-heap stores nodes with finite temporary distance labels, denoted by the set T. A node is added to the heap when it gets a finite temporary distance label and is deleted from the heap when it is permanently labeled. Let $Min(d) = \min \{d(j) : j \in T\}$. The implementation of the R-heap takes advantage of the following two properties that are true for Dijkstra's algorithm.

L1. **Integrality Property.** Each label d(i) is integer.

L2. **Monotonicity of minimum.** *Min(d)* does not decrease following a deletion of a node from the R-heap.

A single level R-heap consists of $1 + K$ subsets of nodes, called *buckets*, for some suitable choice of K. For our implementation of Dijkstra's algorithm, we have $K = 1 + \lceil \log C \rceil$. For $k = 0, 1, \ldots, K$, the nodes in bucket k are denoted by the set CONTENTS(k). We associate with each bucket k a (possibly empty) closed interval of integers $[l_k, u_k]$ which we denote by *range*(k). The l_k and u_k are respectively called the *lower* and *upper limits* of range(k). If $l_k > u_k$, then the range is considered to be empty. We refer to $|\text{range}(k)|$ as the *width* of bucket k. We define the *maximum width* of bucket k to be 2^{k-1}. The ranges of buckets change dynamically throughout the algorithm; nevertheless, for the following discussion, it may be easier to view the ranges as fixed. We store nodes with finite temporary distance labels in the R-heap as follows: Node i will be stored in bucket k if $d(i) \in [l_k, u_k]$. The ranges of the buckets always satisfy the following properties, which makes this assignment uniquely defined.

R1. **Range property.** If a node j has a finite temporary label, then $d(j) \in [l_0, u_K]$.

R2. **Monotonicity Property.** If $j < k$, and if buckets j and k are both nonempty, then $u_j < l_k$. In other words, if $i_1 \in$ CONTENTS(j), $i_2 \in$ CONTENTS(k), then $d(i_1) < d(i_2)$.

R3. **Continuity property.** The union of all ranges is a closed interval $[l_0, u_K]$.

The ranges of the buckets also satisfy the following two additional properties. These properties will be necessary for the algorithm to work; however, their use in the algorithm will not be apparent until much later.

R4. **Redistributivity property.** The width of bucket 0 is 1. The width of the k-th bucket is at most the sum of the maximum width of buckets 0 through k–1, for all $k = 1, \ldots, K$.

R5. **Last bucket property.** $|\text{range}(K)| \geq C$.

In our algorithms, the *initial ranges* of the buckets are as follows:

$\text{range}(0) = [0]$;

$\text{range}(k) = [2^{k-1}, 2^k - 1]$, for $k = 1, \ldots, K$.

Note that the maximum width of bucket K is $2^{\lceil \log C \rceil} \geq C$. For simplicity of exposition, we also create a bucket K+1 which contains all nodes whose temporary distance label is ∞.

We associate with each node i in the heap H an index *assign(i)* which indicates the bucket to which it is assigned.

If $d(i) = \infty$, then $\text{assign}(i) = K + 1$.

If $d(i) < \infty$ and $i \in H$, then $\text{assign}(i) = (k : i \in \text{CONTENTS}(k))$.

We assume that the contents of each bucket are stored as a doubly linked list (see, for example, Aho, Hopcroft and Ullman [1974]). Accordingly, each insertion and deletion from buckets take O(1) steps.

1.2. Dijkstra's Algorithm Using Heap Operations

We perform the following operations on an R-heap.

INITIALIZE(K, H): returns a new empty heap H with buckets $0, \ldots, K + 1$.

REINSERT(i, H) :	reinserts node i in the bucket whose range contains d(i).
DELETE(i, H) :	deletes node i from the heap.
FIND–MIN(i, H) :	returns a node i such that d(i) is minimum among all nodes in H.

The following is a description of Dijkstra's algorithm that uses the above heap operations.

```
algorithm DIJKSTRA;
begin
    INITIALIZE;
    while T ≠ ø do
    begin
        FIND-MIN(i, H);
        T: = T – {i};
        DELETE (i, H);
        for each (i, j) ∈ A(i) do
        begin
            d(j): = min {d(j) , d(i) + c_ij};
            if d(j) = d(i) + c_ij then REINSERT(j, H);
        end;
    end;
end;
```

1.3. The DELETE and REINSERT Operations

The procedures INITIALIZE, DELETE, and REINSERT are all straightforward, and are given below. The procedure FIND-MIN contains the *redistribution* and is deferred to after the numerical example.

```
Procedure INITIALIZE;
begin
      T: = N ;
      d(s): = 0 , and  d(j): = ∞  for all  j ∈  N – {s};
      C: = max{c_ij :  (i, j) ∈  A} ;
      K: = 1 + ⌈log C⌉ ;
      range(0): = { 0 } ;
      for k : = 1 to K do range(k) : = [2^(k-1) ,2^k - 1] ;
      CONTENTS(0): = {0};
      for k : = 1 to K do CONTENTS( k ) : = ø ;
      CONTENTS(K+1): = N – {s} ;
      assign(s): = 0;
      assign(j): = K+1,  for all  j ∈  N – {s};
end;

procedure DELETE(j, H);
begin
      k : = assign(j);
      CONTENTS(k): = CONTENTS(k) – {j} ;
end;


procedure REINSERT(j, H);
begin
      k : = assign(j);
      CONTENTS(k): = CONTENTS(k) – {j} ;
      while d(j) ∉ range(k) do  k : = k – 1;
      CONTENTS(k): = CONTENTS(k) ∪ {j};
      assign(j) : = k ;
end;
```

It follows from properties L2, R1, and R3 that the REINSERT procedure always succeeds in putting the node in an appropriate bucket.

We illustrate the R-heap and the heap operations on the shortest path example given in Figure 1 below. In the figure, the number beside each arc indicates its length.

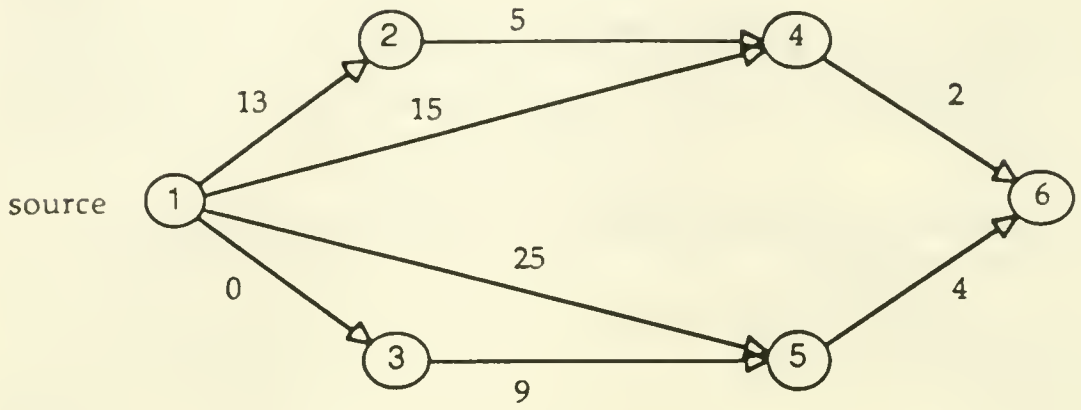

Figure 1. The shortest path example.

For this problem, $C = 25$ and $K = 1 + \lceil \log 25 \rceil = 6$. The starting solution of Dijkstra's algorithm is $d(1) = 0$, and $d(i) = \infty$ for each $i \in \{2, 3, 4, 5, 6\}$. The initial R-heap is presented in Figure 2.

buckets:	0	1	2	3	4	5	6	7
ranges:	[0]	[1]	[2,3]	[4,7]	[8,15]	[16,31]	[32,63]	[∞]
CONTENTS:	{1}	ø	ø	ø	ø	ø	ø	{2, 3, . . . , 6}

i:	1	2	3	4	5	6
d(i):	0	∞	∞	∞	∞	∞
assign(i):	0	7	7	7	7	7

Figure 2. The initial R-heap.

We now illustrate these operations on our shortest path example. In the first iteration, Dijkstra's algorithm selects node 1, deletes it from the R-heap and scans the arcs in $A(1)$ to update the distance labels of adjacent nodes. The new distance labels of nodes 2, 3, 4 and 5 respectively become 13, 0, 15 and 25. The procedure

REINSERT is called to place them in the right buckets. The R-heap at this point appears in Figure 3.

buckets:	0	1	2	3	4	5	6	7
ranges:	[0]	[1]	[2,3]	[4,7]	[8,15]	[16,31]	[32,63]	[∞]
CONTENTS:	{3}	ø	ø	ø	{2,4}	{5}	ø	{6}

i:	2	3	4	5	6
d(i):	13	0	15	25	∞
assign(i)	4	0	4	5	7

Figure 3. The R-heap at the end of Iteration 1.

At this point, the FIND–MIN operation is easy, since bucket 0 contains node 3 which must be the node with the minimum temporary distance label. The algorithm makes node 3 permanent, deletes it from the R–heap, and scans the arc (3, 5) to change the distance label of node 5 from 25 to 9. Since d(5) is decreased, the procedure REINSERT is called, which moves this node to bucket 4. The R–heap now appears as given below.

buckets:	0	1	2	3	4	5	6	7
ranges:	[0]	[1]	[2,3]	[4,7]	[8,15]	[16,31]	[32,63]	[∞]
CONTENTS:	ø	ø	ø	ø	{2,4,5}	ø	ø	{6}

i:	2	4	5	6
d(i):	13	15	9	∞
assign(i)	4	4	4	7

Figure 4. The R-heap at the end of Iteration 2.

Lemma 1. *Let D be the number of calls of the procedure REINSERT. Then the cumulative time spent in all calls to REINSERT is $O(D + nK)$.*

Proof. The time spent in each call of REINSERT is $O(1)$ plus the time spent in the **while** loop. Each repetition of the **while** loop, except the last one, leads to a decrease in the index assign(j) of a node j by one. Thus the total time spent in the **while** loop for reinsertion of any node $j \in N$ is $O(K)$, since assign(j) decreases monotonically from $K+1$ to 0. The cumulative running time, therefore, is $O(D + nK)$. ❑

1.4. The FIND-MIN Operation.

We now consider the FIND–MIN operation. In this operation, we first identify the nonempty bucket with the lowest index. This is easily accomplished in $O(K)$ steps using a sequential search of the buckets. Let bucket p be the lowest-index nonempty bucket. If $p = 0$ or 1, then any node in bucket p has the minimum distance label among all nodes stored in the heap (from properties R4 and R2). If $p \geq 2$, then bucket p contains a node with smallest distance label; but to determine that node we may have to scan the entire contents of bucket p. This time bound would be $O(n)$ in the worst case, and hence totally unsatisfactory for our purposes. However, we can do better. The improvement is based on the following observation: If $p \geq 2$ is the lowest-index nonempty bucket, then it follows from property L2 that the ranges range(0), . . . , range(p – 1) will never be used again for storing temporary labels. We can thus redistribute the range of bucket p into the buckets $0, \ldots, p - 1$ and REINSERT the nodes of CONTENTS(p) in these buckets. As we shall see later, this redistribution of ranges is crucial to improve the overall complexity of our algorithm to $O(m + nK)$. We first illustrate the redistribution on our example and then generalize it subsequently.

In the example given in Figure 4, bucket 4 is the lowest index nonempty bucket. The buckets $0, \ldots, 3$ are empty, and the union of their ranges is $[0, 7]$. The range of bucket 4 is [8, 15], but the smallest distance label in this bucket is 9. By property L2, no temporary distance label will ever be less than 9. We therefore redistribute the range [9, 15] over the lower indexed buckets in the following manner:

range(0) = [9] ,
range(1) = [10] ,
range(2) = [11 , 12] ,
range(3) = [13 , 15] ,
range(4) = ø .

All other ranges do not change. The range of bucket 4 is now empty, and the contents of bucket 4 must be reassigned to buckets 0 through 3. This is accomplished by successively calling the REINSERT procedure. The resulting contents of the buckets are as follows:

CONTENTS(0) = {5} ,
CONTENTS(1) = ø ,
CONTENTS(2) = ø ,
CONTENTS(3) = {2, 4} ,
CONTENTS(4) = ø .

This redistribution necessarily makes bucket 0 nonempty, and the FIND-MIN operation then returns node 5 as the node with smallest distance label.

In general, we redistribute the range of the lowest-index nonempty bucket, say, bucket k with $k \geq 2$, into buckets $0 , \ldots , k - 1$. This procedure consists of modifying the ranges of buckets $0 , \ldots , k$ and then reassigning all the nodes of bucket k. If property R4 is satisfied, then the redistribution of the range of bucket k can always be done and its nodes can be reinserted in the appropriate buckets.

The potential bottleneck operation in the FIND–MIN operation is node scanning. A node is scanned whenever it is in the lowest-index nonempty bucket. In principle, any node j could be scanned in any of the n different FIND–MIN operations; however, in our reinsertion step we are reinserting each scanned node j into a lower index bucket. Thus each node can be scanned at most K different times in all FIND–MIN operations. The total time spent in scanning nodes is O(nK). This is why redistribution of ranges reduces the running time so dramatically.

A formal description of the FIND-MIN operation is given below.

```
procedure FIND-MIN(i, H);
begin
    k: = 0 ;
    while CONTENTS(k) = ø  do k: = k + 1;
    p: = k ;
    if p ∉ {0, 1} then
    begin
        d_min: = min{d(j) : j ∈ CONTENTS(p)};
        range(0) : = { d_min } ;
        let u_p be the upper limit of bucket p;
        if p = K  then
            for  k: = 1 to K  do  range(k): =[ 2^(k-1) + d_min , 2^k + d_min + 1 ] ;
        else
            for  k: = 1 to p  do  range(k):=[ 2^(k-1) + d_min , min(2^k + d_min - 1, u_p)];
        for each j ∈ CONTENTS(p)  do  REINSERT(j, H);
    end;
    if CONTENTS(0) ≠ ø  then return any node i ∈ CONTENTS(0)
    else  return any node i ∈ CONTENTS(1);
end;
```

1.5. Accuracy and Complexity of the Algorithm

Theorem 1. Dijkstra's algorithm implemented using a one-level R-heap determines shortest paths from node s to all other nodes in $O(m + n \log C)$ steps.

Proof. The algorithm is an implementation of Dijkstra's algorithm and to show its correctness it suffices to show that the R-heap correctly stores the temporary labels and correctly determines the minimum temporary distance label. This amounts to showing that the properties R1–R5 are satisfied throughout the algorithm.

Suppose inductively that properties R1–R5 are satisfied at a given step. We first consider an operation in which d(j) is decreased. Decreasing a distance label does not affect the properties R2–R5, but can affect R1. Let node i be the node with

smallest distance label at this iteration. Then $d(i) = l_0$ or $d(i) = l_0 + 1$, depending upon whether node i was in bucket 0 or bucket 1, respectively, prior to its deletion from the R-heap. Note that $u_K \geq l_0 + 1 + C \geq d(i) + c_{ij} = d(j) \geq d(i) \geq l_0$. Hence, $l_0 \leq d(j) \leq u_K$, and property R1 remains satisfied.

Now consider a FIND-MIN step in which the ranges of buckets are modified. Let bucket p be the lowest-index nonempty bucket in this step. If p = K, then the ranges of all buckets are translated from their initial ranges by the amount d_{min}, and by the inductive hypothesis they will satisfy R2–R5. If p < K, then let bucket h be the least-index bucket for which $2^{h-1} + d_{min} > u_p$. Note that h < p, since $u_p = l_p + |range(p)| - 1 < l_p + 2^{p-1} \leq d_{min} + 2^{p-1}$. The new range of each bucket k = 0, 1, . . . , h − 1, is a translation of the initial range by the amount d_{min}; the range of bucket h is $[2^{h-1} + d_{min}, u_p]$; the ranges of buckets h + 1, . . . , p are empty; and other buckets are unaffected. It is easily verified that these new ranges satisfy properties R2–R5.

We next note that modification of ranges in the FIND-MIN step necessarily moves each node in bucket p to a lower index bucket. If p < K, then, as observed above, the new range of bucket p is empty and all of its nodes move to other buckets. Now, consider the case when p = K. If the previous range of bucket K was $[l_K, l_K + 2^K - 1]$, then its new range is $[d_{min} + 2^{K-1}, d_{min} + 2^K - 1]$. Since $d(j) \leq d_{min} + C \leq d_{min} + 2^{K-1}$, during reinsertions all nodes in this bucket move to lower index buckets.

Finally, we analyze the complexity of our algorithm. The DELETE operation is performed n times and each execution requires O(1) time. The REINSERT operation is performed at most m times because of the modifications of distance labels and, by Lemma 1, it requires O(m + n log K) cumulative time. The procedure FIND–MIN is called n times. If either of bucket 0 or 1 is found to be nonempty then this call takes O(1) time. Otherwise, the call requires O(K) time to find the first nonempty bucket and to update the ranges of buckets. This effort is bounded by O(nK) in total. The time needed to find the smallest distance label in bucket p is O(|CONTENTS(p)|). However, subsequently, the REINSERT procedure reinserts each node $j \in$ CONTENTS(p) to a lower index bucket, which can happen at most K times for any node. The total time needed to find the smallest distance labels is thus bounded by O(nK). This also implies that FIND-MIN

operation calls REINSERT operation O(nK) times, requiring O(nK) cumulative time. As $K = 1 + \lceil \log C \rceil$, the theorem follows. ❑

2. The Two-level Redistributive Heap and Dijkstra's Algorithm

In this section, we present a generalized version of the redistributive heap discussed in the previous section which uses a two-level bucket system instead of a one-level bucket system. We show that the two-level bucket system can be used to implement Dijkstra's algorithm in $O(m + n \log C/\log\log C)$ time. We subsequently assume, without any loss of generality that $C \geq 4$, since the shortest path problem with $C \leq 3$ can be solved in O(m) time. Under the similarity assumption, this algorithm runs at least as fast as Fredman and Tarjan's implementation of Dijkstra's algorithm and is asymptotically faster whenever $m/n = O(\log n)$.

2.1. Properties of Two-Level R-heap

The two-level (or two-echelon) R-heap consists of K (big) buckets, and each bucket is further subdivided into L (small) subbuckets. The range of a bucket is the union of the ranges of its subbuckets and, similarly, the contents of a bucket is the union of the contents of its subbuckets. We refer to the h-th subbucket of the bucket k as the *subbucket (k, h)*. An example of an empty two-level R-heap with K = 3 and L = 3 is given in Figure 5.

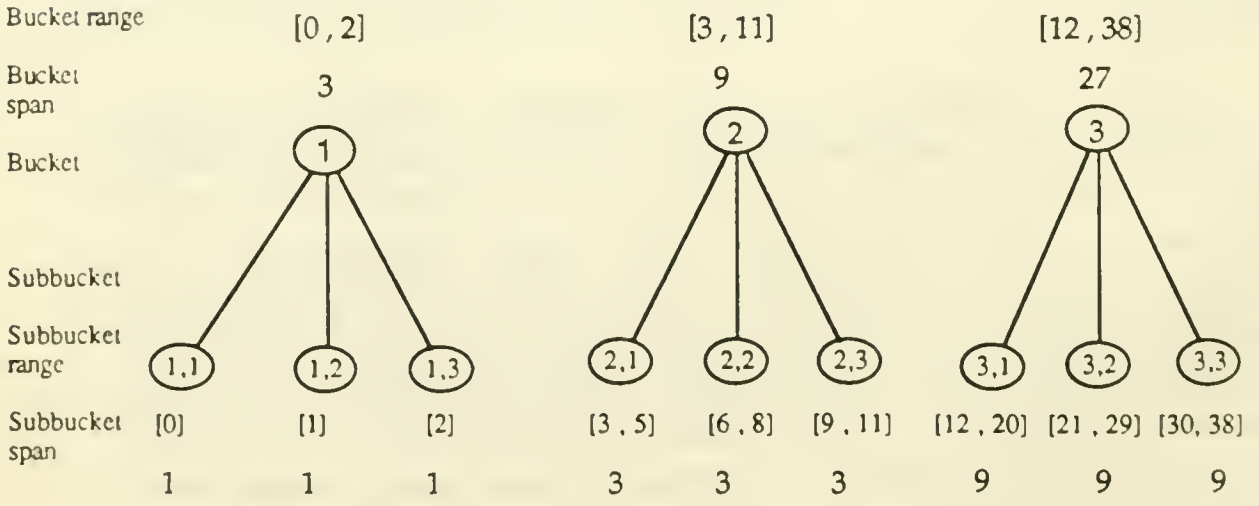

Figure 5. An empty two-level R-heap with K = 3 and L = 3.

In the one-level bucket system, the range of a bucket is redistributed over all of the previous buckets. Thus the maximum width of bucket k is at most the sum of the maximum width of the first $k-1$ buckets. In the two-level bucket system, the range of a *subbucket* is redistributed over all of the previous *buckets*. This allows us to select much larger width of buckets and leads to a reduction in the number of buckets. For example, to store the distance labels in the range [0 , 38], the one-level bucket scheme uses 7 buckets, whereas the two-level bucket scheme given in Figure 5 uses only 3 buckets. Further, once the bucket containing a node is determined,we can determine the appropriate subbucket in O(1) time . This speed of insertion into the subbuckets and the reduction in the number of buckets is translated into an improved complexity bound of the algorithm.

The algorithm suggested in this section also performs multiplication and division by powers of K and L . We assume that both K and L are appropriate powers of 2. This allows us to perform multiplication and division more efficiently, by *shifting* the binary representation of a number. As earlier, the range of the bucket k is a closed interval of integers $[l_k , u_k]$ which we denote by range (k). The maximum width of bucket k is defined to be L^k for all $k = 1, \ldots, K$. The ranges of the buckets satisfy the properties R1, R2, R3, R5, and the following modification of R4.

R4'. **Redistributivity Property.** The width of each subbucket in bucket 1 is 1. The width of a subbucket in bucket k is at most the sum of the maximum width of the buckets 1 through k–1 , for all $k = 2, \ldots, K$.

The initial ranges of the buckets are given below.

$$\text{range}(1) = [0 , L-1];$$
$$\text{range}(2) = [L , L + L^2 - 1];$$
$$\text{range}(3) = [L + L^2 , L + L^2 + L^3 - 1];$$
$$\vdots$$
$$\text{range}(K) = [L + L^2 + \ldots + L^{K-1} , L + L^2 + \ldots + L^{K-1} + L^K],$$

where K and L are positive integers chosen so that $L^{K-1} \geq C$. We show in the Appendix that $K = L = 1 + \lceil 2 \log C / \log \log C \rceil$ satisfies this condition. Clearly, there are many other choices of K and L which also satisfy the above condition, but

setting both equal to $1 + \lceil 2 \log C / \log \log C \rceil$ appears to be a good choice for our data structure. For the sake of convenience, we create a bucket $K + 1$ which contains all nodes with temporary distance label equal to ∞. This bucket has exactly one subbucket. We refer to the above ranges as *initial ranges* in this section. For simplicity, we represent the initial range of bucket k by $[\overset{\circ}{l}_k, \overset{\circ}{u}_k]$.

The range of a bucket is partitioned into the ranges of its subbuckets. The range of a subbucket (k, h) is denoted by the interval $[l_{kh}, u_{kh}]$. The l_{kh} and u_{kh} are respectively called the *lower* and *upper limits* of the subbucket (k, h). The ranges of subbuckets satisfy the following property for every $k = 1, \ldots, K$.

R6. **Subbucket property.**

(i) $\text{range}(k) = \bigcup_{h=1}^{L} \text{range}(k,h)$;

(ii) if $r < h$ and the subbuckets (k, r) and (k, h) are nonempty, then $u_{kr} < l_{kh}$; and

(iii) $|\text{range}(1, h)| = 1$, for each $h = 1, \ldots L$.

We store the ranges of buckets explicitly. The ranges of its subbuckets are maintained implicitly and computed efficiently as needed. Due to the redistribution of ranges, the first few subbuckets of a bucket may be empty. The index e_k represents the number of such empty subbuckets of the bucket k. Initially, $e_k = 0$, for each $k = 1, \ldots, K$. The range of a bucket k is divided over its nonempty subbuckets using the following rule: the first L^{k-1} numbers in the range $[l_k, u_k]$ are given to the subbucket $e_k + 1$, the next L^{k-1} numbers are given to the subbucket $e_k + 2$, and so on. Consequently, the range $[l_{kh}, u_{kh}]$ of a subbucket (k, h) is given by the following expressions:

$$l_{kh} = l_k + (h - e_k - 1)L^{k-1};$$
$$u_{kh} = \min \left\{ l_k + (h - e_k) L^{k-1} - 1, u_k \right\}.$$

If $l_{kh} > u_{kh}$, then the subbucket is considered to be empty. If ranges of the buckets are given by the initial ranges, the range of bucket $k < K$ consists of L^k integer numbers and ranges of its subbuckets consist of L^{k-1} integer numbers in order. The last bucket has all of its subbuckets, except the first one, empty. This is sufficient since the width of a subbucket in bucket K may be as much as L^{K-1}, and

$L^{K-1} \geq C$. It will be seen later that we keep this property satisfied throughout the algorithm.

We store nodes with finite temporary distance labels in appropriate subbuckets. We represent the contents of the subbucket (k, h) by the set CONTENTS (k, h). The contents of a bucket k is the union of the contents of its subbuckets, but we never need this set explicitly. What we need is the number of nodes in a bucket k, which we represent by the cardinality index *card(k)* and maintain throughout the algorithm. Further, we associate with node i in the R-heap a two-tuple set, *assign(i)*, which indicates the bucket and the subbucket to which it belongs. For example, if assign(i) = (k, h), then it implies that node i is in the h-th subbucket of bucket k.

2.2. Heap Operations in Two-Level R-heap

We now describe how the various heap operations needed in Dijkstra's algorithm can be performed on a two-level R-heap. The procedure INITIALIZE given below creates the initial R-heap.

```
procedure INITIALIZE;
begin
    set d(s): = 0 , and d(j): = ∞ for each j ∈ N – {s};
    C: = max {c_ij : (i,j) ∈ A};
    K: = 1 + ⌈2 log C/log log C⌉ ;
    L : = 1 + ⌈2 log C/log log C⌉;
    range(k): =[l°_k, u°_k] , for each k: = 1, . . . , K;

    set e_k : = 0 for each k: = 1, . . . , K;
    CONTENTS (1, 1): = {s};
    CONTENTS(K+1, 1): = N – {s};
    assign(s): = (1, 1);
    assign(i): = (K + 1, 1), for each i ∈ N – {s};
    card(1): = 1;
    card(k): = 0 , for each k = 2, . . . , K;
end;
```

Clearly, the procedure INITIALIZE takes O(n + K) time. The DELETE operation is as easy as in the one-level bucket system, and is given below.

```
procedure DELETE(j, H);
begin
    (k, h): = assign(j);
    CONTENTS(k, h): = CONTENTS(k, h) – {j};
    card(k): = card(k) – 1;
end;
```

The procedure REINSERT(j, H) is also similar. If node j is to be moved from its present subbucket, then by scanning the buckets we determine the bucket whose range includes d(j). Then, in O(1) time we determine its subbucket whose range includes d(j) and add node j to its contents. A formal description of this procedure is given below.

```
procedure REINSERT(j, H);
begin
    (k, h): = assign(j);
    if d(j) ∉ range(k, h) then
    begin
        CONTENTS(k, h): = CONTENTS(k, h) – {j};
        card(k): = card(k) – 1;
        while d(j) ∉ range(k) do k: = k – 1;
        r: = ⌊(d(j) – l_k)/L^(k–1)⌋ + e_k + 1
        assign(j): = (k, r);
        CONTENTS(k, r): = CONTENTS(k, r) + {j};
        card(k): = card(k) + 1;
    end;
end;
```

Lemma 2. In the two-level R-heap , D calls of the procedure REINSERT take O(D + nK) total time.

Proof. Each execution of the REINSERT procedure takes O(1) time plus the time spent in the **while** loop. Each iteration of the **while** loop moves a node to a smaller

index bucket. As there are K buckets, D calls of this procedure would take O(D + nK) overall time. ■

The procedure FIND-MIN is described below, followed by its explanation.

```
procedure FIND-MIN(i, H);
begin
    k: = 1;
    while card(k) = 0 do k: = k + 1;
    p: = k and h: = e_k + 1 ;
    while CONTENTS(p, h) = ø do h: = h + 1;
    if p > 1 then
    begin
        d_min: = min{d(j): j ∈ CONTENTS(p, h)};
        if p = K then
            for k: = 1 to p do
                range(k): = [l°_p + d_min , u°_p + d_min] and e_k: = 0;
        else
        begin
            let u_ph be the upper limit of the subbucket (p, h);
            for k: = 1 to p − 1 do
                range(k): = [l°_p + d_min , min {u°_p + d_min, u_ph}] and e_k: = 0;
                range(p): = [ u_ph + 1 , u_p] and e_p: = h;
        end;
        for each j ∈ CONTENTS(p, h) do REINSERT(j, H) ;
        h: = 1;
    end;
    return any node i ∈ CONTENTS(1, h);
end;
```

The procedure FIND-MIN proceeds by determining the lowest-index nonempty bucket p. Next, by scanning its subbuckets, the lowest-index nonempty subbucket (p, h) is identified. These two steps require O(K) and O(L) time respectively. If p = 1 , then every node in the subbucket (p, h) has the smallest

distance label (by property R6 (iii)). If $p > 1$, then ranges of buckets are modified and nodes in the subbucket (p, h) are reinserted to lower index buckets. This analysis is divided into two cases. If $p = K$, then the new ranges of buckets are the initial ranges translated by the amount d_{min}. As in Theorem 1, it can be shown that this keeps the properties R1, R2, R3, R4', and R5 satisfied, moves the nodes with smallest distance labels to the subbucket (1, 1), and makes the new range of bucket K completely disjoint from its previous range. If $p < K$, then the range of the subbucket (p, h) is redistributed over the buckets 1 to $p - 1$; the range of bucket p is modified as $(u_{ph} + 1, u_p)$; and the first h subbuckets of the bucket p are made empty. In either case, all nodes in the subbucket (p, h) move to the lower index buckets during reinsertions. Since there are K buckets, each node is scanned O(K) times in all FIND-MIN steps. The REINSERT procedure is called O(m + nK) times and it takes equal amounts of time to execute all the calls. Since $K = L = \lceil 2 \log C / \log \log C \rceil + 1$, we get the following result:

Theorem 2. Dijkstra's algorithm implemented using a two-level R-heap runs in O(m + n log C/log log C) time. ■

Finally ,we discuss the time needed to initialize the subbuckets as empty sets. This initialization takes O(KL) time, which could dominate the running time if C is exponentially large. In such a case, the semi-logarithmic model of computation is more appropriate as discussed in Section 5. Nevertheless, we can reduce the initialization time to O(nL) by postponing any insertion of a node into subbuckets of bucket k until k is selected for the first time in a FIND-MIN operation. In this way, the initialization time is dominated by the running time of the algorithm.

3. A Further Improvement if $C < n$

In this section, we suggest an improved version of the algorithm described in Section 2. The improved algorithm runs in $O(m + n \log C / \log \log nC)$ time. This improvement is obtained by using larger numbers of subbuckets with each bucket and using a more efficient technique to locate the first nonempty subbucket of a bucket.

This data structure consists of K buckets and each bucket has $K \lfloor \log n \rfloor$ subbuckets. The value of K is chosen so that $(K \lfloor \log n \rfloor)^{K-1} \geq C$. It can be easily shown that the value of $K = \lceil \log C / \max(\log \log C, \log \log n) \rceil$ satisfies the above

condition, and log C/max (log log C, log log n) = O(log C/log log nC). Using more subbuckets with each bucket does not affect the reinsertion time since it depends solely on the number of buckets. However, the FIND-MIN step has to be performed more cleverly as it involves sequentially scanning the subbuckets of a bucket to find the least index nonempty subbucket. We use the following technique to speed up this operation. We partition the subbuckets of a bucket into K groups of $\lfloor \log n \rfloor$ subbuckets each. We associate with each such group a binary number of $\lfloor \log n \rfloor$ bits, whose i-th bit is 1 if the i-th subbucket in the group is non-empty, and 0 otherwise. We refer to this number as the *binary number* of that group. Thus the binary number is an integer no more than n. In the FIND-MIN step, we first scan through the buckets to identify the least-index nonempty bucket. We then scan through its groups of subbuckets, in order, to identify the first group whose binary number is nonzero. Both of these steps take O(K) time. Next, we identify the first non-zero bit in the selected group by a *table-lookup*. This consists of preparing a table at the beginning of the algorithm consisting of values (x, y) where for all $1 \leq x \leq n$ the corresponding y value denotes the first nonzero bit in the number x. For a given group, we use its binary number (as x) in the table to find its first nonzero bit (as y) in O(1) time. Further, the binary numbers of groups of subbuckets can be easily maintained with an additional overhead of O(1) operations per DELETE and REINSERT operation. Consequently, the above algorithm runs in O(m + nK) = O(m + n log C/log log nC) time.

4. Implementation Using Fibonacci Heaps

We show in this section that the two-level bucket system can be implemented in $O(m + n\sqrt{\log C})$ time using a variant of Fibonacci heaps. Observe that the two-level method requires O(m + nK) time except for the step of finding the first nonempty subbucket which takes O(L) time for each FIND-MIN step. Our improvement results from having a much larger number of subbuckets for each bucket, i.e., $K << L$, and using Fibonacci heaps to select the first nonempty subbucket of a bucket. For this purpose, we number the subbuckets (of all the buckets) consecutively 1 through M = LK and associate with each node the index of the subbucket to which it belongs. This index is called the *key* of the node. It may be pointed out that the data structure we describe now is *in addition to* the two-level bucket system described in the previous section and its sole purpose is to identify a

node with smallest key which is equivalent to finding the least-index nonempty subbucket in the above data structure.

The Fibonacci heap (abbreviated as F-heap) data structure is able to perform the following operations efficiently:

find-min:	Find and return a node of minimum key.
insert(x):	Insert a new node x with predefined key into a collection of nodes.
decrease (value, x):	Reduce the key of node x from its current value to *value*, which must be smaller than the key it is replacing.
delete(x):	Delete node x from the collection of nodes.

The Fibonacci heaps of Fredman and Tarjan [1984] support these operations in the following *amortized time bounds* (By amortized time we mean the time per operation averaged over a worst-case sequence of operations. For a thorough discussion of this concept, see Tarjan [1985] and Mehlhorn [1984]) : O(1) for *find-min*, *insert*, and *decrease*, and O(log k) for *delete*, where k is the heap size. These bounds are also attained by *relaxed heaps* due to Driscoll et. al [1987] and *V-heaps* due to Peterson [1987]. We are interested in the case in which n can be much larger than M, i.e. many items have the same key. We show below how to modify F-heaps so that the amortized time per delete is reduced to O(log min {n, M}) without changing the O(1) amortized time bound for the other three operations. For the two level bucket system, we select $L = 2^{K-1}$ and $K = \lceil \sqrt{\log C} \rceil + 1$. Note that this choice of L and K assures that $L^{K-1} \geq C$ which is a necessary condition for the two level R-heap. There are a total of n *insert* operations, n *find-min* operations, O(m + nK) *decrease* operations and n *delete* operations in the algorithm. Using the above data structure, these operations take a total of $O(m + nK + n \log (LK)) = O(m + n\sqrt{\log C})$ time, since $\log(LK) = K-1 + \log K = O(\sqrt{\log C})$.

We now discuss the modification of the F-heaps to show that the amortized time per *delete* operation can be reduced to O(log min {n, M}) without changing the O(1) amortized time bound for the other operations. The main idea is to make sure that the F-heap contains at most M items, i.e., 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 heaps. A heap-ordered tree is a rooted tree such that if p(x) is the parent of node x, key(p(x)) ≤ key(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 with the smaller key the parent of the root with the larger key, breaking a tie arbitrarily. A link operation takes O(1) time. In an F-heap, a pointer is maintained to a tree root of smallest key, making *find-min* an O(1) time operation. An insertion consists of creating a new one-node tree and adding it to the collection of trees. This also takes O(1) time.

Each non-root node in an F-heap is in one of the two states, *marked* or *unmarked*. When a node becomes a non-root by losing a comparison during a link, it becomes unmarked. A *decrease* operation on a node x is performed as follows. First the value of x is updated. Then, if x is not already a tree root, the arc joining x and its parent p(x) is cut and the following step is repeated, with y initially equal to (the old) p(x), until y is unmarked or y is a tree root, cut the arc joining y and its parent p(y), and replace y by (the old) p(y). After the last such cut, if the last node y is not a root, it is marked. The overall effect of a *decrease* operation is to break the initial tree containing x into possibly several trees, one of which is rooted at x. The time required by the *decrease* operation is O(1) plus O(1) per cut. Since at most one node is marked per cut, 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 heap operations is at most twice the number of *decrease* operations, even though a single *decrease* can result in many cuts.

The fourth heap operation, *delete(x)*, is done by first performing decrease(key (x), x), which does not affect key (x) but makes x a tree root, then removing node x, making each of its children a tree root, and finally repeatedly linking trees having roots of equal rank until no two tree roots have equal rank.

Fredman and Tarjan showed that when rooted trees are manipulated in the ways described above, the following invariant is maintained: for any node x, rank (x)

= O(log d(x)), where *rank(x)* is the number of children of x and *d(x)* is the number of *descendants* of x. A simple analysis gives an amortized time bound of O(1) for *find-min*, *insert*, and *decrease*, and O(log n) for *delete*.

Now we are ready to solve the problem posed earlier. For each value $i \in [1, M]$, we maintain the set S(i) of items x with key(x) = i. One item in S(i) is designated as the *representative* of S(i). All the items, both the representatives and the non-representatives, are grouped into heap-ordered trees of the kind manipulated by the F-heap algorithm. These trees are divided into two groups: *active trees*, whose roots are representatives, and *passive trees*, whose roots are non-representatives. The following invariant that every non-representative is a root of the heap ordered tree and that key (p(x)) < key(x) is maintained throughout the algorithm.

We note the following important features of this representation: the representative of the nonempty set S(i) of minimum i is the root of an active tree, and the total number of nodes in active trees is at most M. (Every node in an active tree is a representative.) To facilitate *find–min*, we maintain a pointer to the active tree root of minimum key; thus *find–min* takes O(1) time. We perform *insert(x)* by making x 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 so, x becomes the representative of S(i). We perform *decrease* (value, x) as described above, breaking the tree containing x into one or more new trees, with the following change: x is moved from S(key(x)). If x was a representative then some other item in S(key(x)) (if any) is made its new representative, and the tree rooted at the new representative becomes active. Further, if S(value) is empty, then x becomes a representative. Otherwise, it remains a non-representative. Other new roots created by the cuts during the decrease become the roots of active trees. The total time required by the decrease is O(1) plus O(1) per cut, including the time necessary to move trees between the active and passive groups.

We perform the *delete* operation as described above, except that the repeated linking is performed only on active trees. That is, active trees of equal rank are repeatedly linked until there is at most one active tree per rank.

We analyse the efficiency of the four heap operations in almost exactly the same way as in Fredman and Tarjan [1984]. We define the potential of a collection of

rooted trees to be the number of trees plus twice the number of marked nodes they contain. We 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 and the potential is always nonnegative. Thus for any sequence of heap operations the total amortized time is an upper bound on the total actual time.

The amortized time of a *find–min* is O(1), since it does not change the potential. An *insert* increases the potential by one and thus also has an O(1) amortized time bound. A *decrease* operation resulting in k cuts adds O(1) – k to the potential (each cut, except for at most one adds a tree but removes a marked node). Thus a *decrease* takes O(1) amortized time if we regard a cut as taking unit time.

Each link during a *delete* reduces the potential by one and thus has an amortized time of zero, if we regard a link as taking unit time. The additional time spent during a cut is O(log min {n, M}) , as is the potential increase caused by removing a node of minimum key, since the maximum rank of an active tree is O(log min {n, M}). Thus the amortized time of a *delete* is O(log min {n, M}), as desired. Finally, the cost of creating empty sets S(i) is O(M). We have thus shown the following result:

Theorem 3. Dijkstra's algorithm implemented using the two-level R-heap and the modified Fibonacci heaps runs in $O(m+ n \sqrt{\log C})$ *time.* ■

The idea used here, that of grouping trees into active and passive trees, applies as well as to V-heaps to give the same time bounds, but it does not seem to apply to relaxed heaps.

5. The Semi-Logarithmic Model of Computation

In the previous sections, we analysed our algorithms in the uniform model of computation. In particular, we assumed that arithmetic on integers in the range [0 , nC] has cost O(1). In this section, we analyse our algorithms under the semi-logarithmic model of computation. The bottleneck operation in the straightforward analysis of our algorithms in the semi-logarithmic model of computation is the comparison whether d(j) ∈ range(k) during reinsertions. We slightly vary the algorithm so that this step can be performed by looking at a *small* number of bits of d(j) and we obtain the improved time bounds.

The exact definition of the semi-logarithmic model is given by the following two assumptions:

A1. Arithmetic and all other RAM-operations (index calculations, pointer assignments, etc.) on integers of length O(log n) have cost O(1), and that

A2. $\log C = O(n^{O(1)})$. (Alternatively, $C = O(2^{n^{O(1)}})$.

Remark: In this model, we store numbers larger than O(log n) in more than one word. We represent arc lengths and distance labels as arrays of length $\lceil (\log nC)/R \rceil$, where $R = \lfloor (\log n)/2 \rfloor$. Each array element is an integer in the range $[0, 2^R - 1]$. Since we assume (assumption A1) that indexing into these arrays has cost O(1), and this is only reasonable if the indexes are O(log n) in length we are forced to the assumption that $\lceil \log nC/\log n \rceil = O(n^{O(1)})$; which justifies assumption A2. ■

Let us first analyse the basic algorithm of Section 1. We consider the following alternative description of the algorithm. The accuracy of this version can be proved similarly to that of the original algorithm, and therefore its proof is omitted. Let $K = \lceil \log nC \rceil$, let *Min (d)* be the minimum finite temporary distance label, and let $\alpha_{K-1} \ldots \alpha_0$ be the binary representation of *Min (d)*, i.e., $\alpha_i \in \{0, 1\}$ and *Min (d)* = $\sum_{i=0}^{K-1} \alpha_i 2^i$. As before, we have buckets $0, \ldots, K+1$ with

$i \in$ CONTENTS(0) iff $d(i) =$ *Min (d)*,
$i \in$ CONTENTS(k) iff *Min (d)* $< d(i) < \infty$, $\beta_{K-1} \ldots \beta_0$ is the binary representation of d(i) and k is the maximal index for with $\beta_{k-1} \neq \alpha_{k-1}$, and
$i \in$ CONTENTS(K + 1) iff $d(i) = \infty$.

Our algorithm requires successive bits in the binary representation of d(j). Let *bit*(j, k) denote the k-th bit in the binary representation of d(j). We show that bit(j, k) can be computed in O(1) time. As already indicated, we store a temporary distance label d(j) as an array d(j, ·) of length $\lceil \log nC/R \rceil$ of number of R bits each. The index k = assign(j) for each j is stored as a pair (k_1, k_2) with $0 \leq k_2 < R$ and $k = k_1 R + k_2$. Then bit(j, k) is the k_2-th bit in the number $d(j, k_1)$. We can calculate this bit by using the table BIT, where BIT(v, j) is the j-th bit in the binary expansion of v for each $v \in [0, 2^R - 1]$ and each $j \in [0, R-1]$. The table BIT is easily computed in linear time.

Then the k_2-th bit in the number $d(j, k_1)$ is given by $BIT(d(j, k_1), k_2)$ and it can be computed in O(1) by a table look-up in the table BIT.

With these definitions, procedures REINSERT and FIND-MIN can be reformulated as follows:

```
procedure REINSERT (j, H);
begin
    k : = assign(j);
    CONTENTS(k) : = CONTENTS(k) – {j};
    while bit (j, k) = α_k and k > 0 do k : = k – 1;
    CONTENTS(k) : = CONTENTS(k) ∪ {j};
    assign(j) : = k
end;

procedure FIND-MIN(i, H);
begin
    while card(k)= 0 do k : = k + 1;
    Min(d) ← min{d( j) : j ∈ CONTENTS(k)};
    let α_{k-1} . . . α_0 be the binary expansion of Min(d);
    for all j ∈ CONTENTS(k) do REINSERT(j, H);
end;
```

A call of FIND-MIN has cost O(log nC) exclusive of the cost of the calls to REINSERT. The total cost of the calls to REINSERT is O(m + n log nC). Finally, the cost of relabeling is $O(\lceil \log nC/\log n \rceil)$ per relabeling step and hence $O(m\lceil \log nC/\log n \rceil)$ overall. We summarize in:

Theorem 4. *Under the assumptions A1 and A2, the modification of the algorithm of Section 1 runs in time* $O(m\lceil \log nC/\log n \rceil + n \log nC)$. ■

It is worthwhile to compare this time with the running time of Fredman and Tarjan's algorithm under this model. In Fibonacci-heaps every step involves the comparison between two keys and hence has cost $O(\lceil \log M/\log n \rceil)$ when the keys are at most M. Thus Dijkstra's algorithm with Fibonacci-heaps has running time $O(m\lceil \log nC/\log n \rceil + n \log n\lceil \log nC/\log n \rceil)$, which agrees with the time bound of Theorem 4. (There does not appear to be any simple modification of the Fibonacci

heap data structure to improve running times under the semi-logarithmic model.) However, the present implementation is simpler and has smaller constant factors. It is also worth noting that it takes time $O(m\lceil \log C/\log n\rceil)$ to read the input in our model. Thus the modification of the algorithm of Section 1 is optimal whenever $m \geq n \log n$.

We turn to the algorithms of Sections 2 and 4 next. In these algorithms each bucket is represented as an array of L subbuckets. An index into such an array has log L bits. In the spirit of assumption A1 we restrict ourselves to log L = O(log n). The assignment of nodes to buckets is as follows: Let $K = \lceil \log nC/\log L\rceil$ and let $\alpha_{K-1} \ldots \alpha_0$ be the L-ary expansion of *Min(d)*, i.e., $0 \leq \alpha_i < L$ and $Min(d) = \sum \alpha_i L^i$. Then a node $i \in T$ with $d(i) < \infty$ belongs to subbucket(k, h) if either *Min(d)* = d(i) and k = 0, $h = a_0$; or if *Min(d)* < d(i), $\beta_{k-1} \ldots \beta_0$ is the L-ary representation of d(i), k is the maximal index with $\beta_{k-1} \neq \alpha_{k-1}$ and $h = \beta_{k-1}$. Note that the buckets are numbered 0 through K–1 and for each bucket, subbuckets are numbered 0 through L–1.

As above, we store label d(j) as an array d(j, ·) where each array element is a $\lfloor(\log n)/2\rfloor$ bit number. We assume that log L divides $\lfloor(\log n)/2\rfloor$. Then the k-th digit in the L-ary representation of d(j), represented by DIGIT(j, k), can be computed in time O(1) in the same way as we computed the k-th digit in the binary expansion above. The procedure REINSERT is now given below.

```
procedure REINSERT(j, H);
begin
    (k, h) := assign(j);
    CONTENTS(k, h): = CONTENTS(k, h) – {j};
    card(k) : = card(k) – 1;
    while DIGIT(j, k) = α_k and k > 1 do k : = k – 1;
    CONTENTS(k, DIGIT(j, k)) : = CONTENTS(k, DIGIT(j, k)) ∪ {j};
    card(k) : = card(k) + 1;
end;
```

The total cost of all calls to REINSERT is $O(m + nK)$. Also, the total cost of arithmetic in the relabel step is $O(m\lceil \log nC/\log n \rceil)$. The procedure FIND-MIN is given below.

```
procedure FIND-MIN(i, H);
begin
    k := 0;
    while card(k) = 0 do k := k + 1;
    p := k and h := 0;
    while CONTENTS(k, h) = ø do h := h + 1;
    if p > 0 then
    begin
        Min(d): = min{d(j) : j ∈ CONTENTS(p, h)};
        let α_{K-1} . . . . .α_0 be the L-ary expansion of Min(d);
        for all j ∈ CONTENTS(p, h) do REINSERT(j, H);
        h := α_0;
    end;
    return any node i in CONTENTS(0, h);
end;
```

A call to FIND-MIN has cost $O(K + L)$ exclusive of the costs of reinserting. Thus the total cost of the shortest path algorithm is $O(m\lceil \log nC/\log n \rceil + n(K + L))$.

We select $K = \lceil \log nC/\log n \rceil$. If $\log nC/\log\log nC < 2^{\lfloor (\log n)/2 \rfloor}$, then we select L to be the greatest power of 2 which is less than or equal to $\log nC/\log\log nC$ and for which value $\log L$ divides $\lfloor (\log n)/2 \rfloor$; otherwise L is set to $2^{\lfloor (\log n)/2 \rfloor}$. In either case, the running time of the algorithm reduces to $O(m\lceil \log nC/\log n \rceil + n \log nC/\log\log nC)$. We thus obtain the following result:

Theorem 5. The algorithm of Section 2 runs in time $O(m\lceil \log nC/\log n \rceil + n \log nC/\log\log nC)$. ■

Finally, for the algorithm in Section 4 we use Fibonacci-heaps with the domain $[0, KL]$. Thus the total cost of all calls to REINSERT increases to $O((m + nK)\lceil \log KL/\log n \rceil)$ since each of the $m + nK$ calls to REINSERT brings about an additional cost of $O(\lceil \log KL/\log n \rceil)$ for the *decrease* operation in the Fibonacci-heap. Further, each call of *find-min* has cost $O(\log KL \lceil \log KL/\log n \rceil)$.

Since a *find-min* in a Fibonacci-heap requires O(log KL) steps and each step has a cost of $O(\lceil \log KL/\log n \rceil)$ by the remark following Theorem 4. Thus the total cost is $O(m(\lceil \log nC/\log n \rceil + \lceil \log KL/\log n \rceil) + n(K + \log KL)\lceil \log KL/\log n \rceil)$. We select $K = \lceil \sqrt{\log nC} \rceil$ and $\log L = \lceil \sqrt{\log nC} \rceil$ for $\log nC \le (\log n)^2$; and $K = \log nC/\log n$ and $\log L = \lfloor (\log n)/2 \rfloor$ for $\log nC \ge (\log n)^2$. It can be easily verified that in both the cases, the above expression reduces to $O(m\lceil \log nC/\log n \rceil + n\sqrt{\log nC})$ We thus obtain the following theorem.

Theorem 6. The algorithm of Section 4 runs in time $O(m\lceil \log nC/\log n \rceil + n\sqrt{\log nC})$. ■

We conclude that the algorithm of Section 4 is optimal whenever $\log nC \ge (\log n)^2$, i.e., $C \ge n^{\log n}$, or if $m \ge n \log n/\sqrt{\log nC}$.

6. Conclusions and Research Issues

We have presented several new efficient algorithms for the shortest path problem, all based on a new data structure which we call a redistributive heap. The simplest two-level version of this algorithm runs in $O(m + n \log C/ \log \log C)$ time using the uniform model of computation, which improves on previous algorithms under quite reasonable assumptions on the data. In addition, it is sufficiently simple to be very efficient in practice. Preliminary testing indicates that it is comparable to the best other implementation of Dijkstra's algorithm. A more complex version of the algorithm uses Fibonacci heaps and runs in time $O(m + n\sqrt{\log C})$. Although this algorithm is not as efficient in practice, but it is asymptotically better in the worst case.

In addition to the results mentioned above, we have also considered algorithms under the semi-logarithmic model of computation, and we have modified our algorithms appropriately. In the case when C does not satisfy the similarity assumption, the semi-logarithmic model better reflects current sequential computation than does the uniform model. In addition to this added degree of realism, the semi-logarithmic model also avoids an implicit restriction of the uniform model, viz., the semi-logarithmic model allows the user to operate on a small part of a large number without getting *charged* for using the whole number. In the uniform model of computation, each arithmetic step counts as one operation, and so there is no incentive to perform arithmetic operations more efficiently. In the semi-logarithmic model of computation, we count operations more accurately to

reflect that some arithmetic operations, such as multiplication by 2 or determining whether an integer is odd, are inherently faster than other operations, such as multiplication by 3.

This last degree of flexibility in the semi-logarithmic model is best reflected in the fact that our algorithms are linear time in the data when $\log C = \Omega(n)$. Under the uniform model of computation, our algorithms appear increasingly worse as C gets large. In reality, if C gets sufficiently large, then the time to solve the problem is proportional to the time to read the data. This improved relative efficiency of the algorithm for large C is not and can not be captured by the uniform model of computation.

Currently, under the semi-logarithmic model of computation, our algorithms are linear time for a wide range of parameter choices. It is an open question as to whether the algorithm can be made linear time for all possible inputs. It appears that the most difficult case is when C satisfies the similarity assumption.

The redistributive heap data structure is not easily generalized to solve other problems because of the assumption that the minimum in the FIND-MIN step is monotonically nondecreasing. It would be interesting to know whether this assumption can be relaxed, or whether the data structure can be used to solve other problems as well.

Acknowledgements

We thank Hershel Safer for a careful reading of the paper and many useful suggestions. The research of the first and third authors was supported in part by the Presidential Young Investigator Grant 8451517-ECS of the National Science Foundation, by Grant AFORS-88-0088 from the Air Force Office of Scientific Research, and Grants from Analog Devices, Apple Computer, Inc. and Prime Computer. The research of the second author was supported by Grant DFG Sonderforschungsbereich 124, TPB2. The research of the fourth author was partially supported by National Science Foundation Grant No. DCR-8605962 and Office of Naval Research Contract No. N00014-87-0467.

Appendix.

Lemma 1. $1 + \frac{1}{2} \log \log C \geq \log \log \log C$, *for all values of* $C \geq 1$.

Proof. We prove this result by simple case analysis.

Case 1. $1 \leq C < 2^4$. For these values of C, $\log \log \log C < 1$ and the inequality holds.

Case 2. $2^4 \leq C < 2^{16}$. For these values of C, $\frac{1}{2} \log \log C \geq 1$ and $\log \log \log C < 2$. Hence the inequality still holds.

Case 3. $C \geq 2^{16}$. For $C = 2^{16}$, $\frac{1}{2} \log \log C = \log \log \log C = 2$. For all higher values of C, $\frac{1}{2} \log \log C$ dominates $\log \log \log C$. ❐

Lemma 2. *Let* $L = \frac{2 \log C}{\log \log C}$. *Then* $L^L \geq C$ *for all values of* $C \geq 1$.

Proof. Showing $L^L \geq C$ is equivalent to showing that $L \log L \geq \log C$. Substituting $L = \frac{2 \log C}{\log \log C}$ in the left-hand side, we get

$$\begin{aligned} L \log L &= (2 \log C / \log \log C) \log (2 \log C / \log \log C) \\ &= (2 \log C / \log \log C) [1 + \log \log C - \log \log \log C] \\ &\geq (2 \log C / \log \log C) \left(\frac{1}{2} \log \log C \right) \quad \text{(by Lemma 1).} \end{aligned}$$

Therefore, $L \log L \geq \log C$. ❐

REFERENCES

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

Boas,P. Van Emde, R. Kaas, and E. Zijstra. 1977. Design and Implementation of an Efficient Priority Queue. *Math. Sys. Theory* 10, 99-127.

Denardo, E.V., and B. L. Fox. 1979. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.* 27, 161–186.

Dial, R. 1969. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM* 12, 632-633.

Dial, R., F. Glover, D. Karney, and D. Klingman. 1979. A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. *Networks* 9 , 215–248.

Dijkstra, E. 1959. A Note on Two Problems in Connexion with Graphs. *Numeriche Mathematics* 1, 269–271.

Driscoll, J. R., H. N. Gabow, R. Shrairman, and R. E. Tarjan. 1987. Relaxed Heaps: An Alternative to Fibonacci Heaps, Technical Report, CS-TR-109-87, Department of Computer Science, Princeton University, Princton, NJ, submitted to *Comm. ACM.*

Fredman, M.L., and R. E. Tarjan. 1984. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *25th Annual IEEE Symp. on Found. of Comp. Sci.,* 338–346, also in *J. of ACM* 34(1987), 596–615.

Gabow, H.N. 1985. Scaling Algorithms for Network Problems. *J. of Comp. and Sys. Sci.* 31 , 148–168.

Gilsinn, J., and C. Witzgall. 1973. A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees. Technical Note 772, National Bureau of Standards, Washington, D.C.

Glover, F., D. D. Klingman, N. V. Phillips, and R. F. Schneider. 1985. New Polynomial Shortest Path Algorithms and Their Computational Atttributes. *Man. Sci.* 31, 1106–1128.

Johnson, D. B. 1977a. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* 24, 1–13.

Johnson, D. B. 1977b. Efficient Special Purpose Priority Queues. *Proc. 15th Annual Allerton Conference on Comm., Control and Computing*, 1–7.

Johnson, D. B. 1982. A Priority Queue in Which Initialization and Queue Operations Take O(log log D) Time. *Math. Sys. Theory* 15, 295–309.

Mehlhorn, K. 1984. *Data Structures and Algorithms*. Vol. 1, Springer Verlag.

Pape, U. 1980. Algorithm 562: Shortest Path Lengths. *ACM Trans. Math. Software* 6, 450–455.

Peterson, G. L. 1987. A Balanced Tree Scheme for Meldable Heaps with Updates. Technical Report, GIT-ICS-87-23, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

Tarjan, R. E. 1985. Amortized Computational Complexity. *SIAM J. Alg. Disc. Math.* 6 306–318.

Williams, J .W. J. 1964. Algorithm 232: Heapsort. *Comm. ACM* 7, 347–348.