Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation

JAMES R. DRISCOLL, HAROLD N. GABOW, RUTH SHRAIRMAN, and ROBERT E. TARJAN

ABSTRACT: The relaxed heap is a priority queue data structure that achieves the same amortized time bounds as the Fibonacci heap—a sequence of $m$ decrease_key and $n$ delete_min operations takes time $O(m + n \log n)$. A variant of relaxed heaps achieves similar bounds in the worst case—$O(1)$ time for decrease_key and $O(\log n)$ for delete_min. Relaxed heaps give a processor-efficient parallel implementation of Dijkstra's shortest path algorithm, and hence other algorithms in network optimization. A relaxed heap is a type of binomial queue that allows heap order to be violated.

1. INTRODUCTION
The Fibonacci heap data structure of Fredman and Tarjan allows an optimal implementation of Dijkstra's shortest path algorithm [3]. It is central to the best-known algorithm for minimum spanning trees [5] and many other algorithms. These applications are based on the fact that, with a Fibonacci heap, a sequence of $m$ decrease_key and $n$ delete_min operations takes time $O(m + n \log n)$. Equivalently, Fibonacci heaps achieve an amortized time of $O(1)$ for decrease_key and $O(\log n)$ for delete_min.

This article presents a new data structure called the relaxed heap. Two implementations of relaxed heaps are given. The first achieves the same amortized bounds as Fibonacci heaps, but maintains more structure. This structure may make relaxed heaps faster in practice. The second implementation of relaxed heaps gives a theoretical improvement over Fibonacci heaps: it achieves the above time bounds for decrease_key and delete_min in the worst case, rather than in the amortized case. The article concludes with some applications. The main result is a processor-efficient parallel algorithm for shortest paths on a directed graph with nonnegative edge lengths. Specifically, for an EREW PRAM parallel machine with $p$ processors and a graph with $n$ vertices and $m \geq n \log n$ edges, an optimal running time of $O(m/p)$ is achieved for $p = m/(n \log n)$. This result can be achieved using either implementation of relaxed heaps. We do not know whether the same result can be achieved using Fibonacci heaps. The parallel shortest path algorithm can be used as a subroutine to get processor-efficient algorithms for other problems in network optimization, such as the minimum spanning tree problem, assignment problem, transportation problem, and others.

Relaxed heaps are based on a more structured family of trees than Fibonacci heaps, namely the binomial trees. The height of a binomial tree of $n$ nodes is a factor $\log_2 \phi = 0.69+\times$ the height of a Fibonacci tree. This improves the constant in asymptotic estimates. Whether these savings are actually realized in
practice is a matter for experimental verification, which we have not done. (Brown [2] shows that the closely related binomial queues are efficient in practice.) Relaxed heaps give a large family of alternatives to Fibonacci heaps. These alternatives provide both theoretical insight and possibly the flexibility needed for efficient practical implementation.

 Concurrent with this work, Peterson [12] has proposed an elegant data structure that achieves the same amortized bounds as Fibonacci heaps. It has a number of interesting properties, such as working with binary trees. Comparing Peterson's work with the concerns of this article, [12] does not achieve the bounds in the worst case, and does not seem to support the applications to parallel computation given here.

 In this article log n denotes logarithm to the base two. A priority queue is a data structure for storing a set of items x, each having a numerical key denoted k(x). The main operations are:

- **make-heap**: initialize a heap to store the empty set;
- **insert(x)**: make x a new item in the heap;
- **decrease-key(x, v)**: decrease key k(x) to a smaller value v;
- **delete-min(x)**: delete an item of minimum key from the heap and return it as x.

Two useful additional operations are:

- **find-min(x)**: return an item of minimum key as x;
- **delete(x)**: delete item x from the heap.

Other operations will be introduced as needed.

Binomial queues were introduced by Vuillemin [14]. The binomial trees B_r are defined recursively as follows: B_0 is one node, B_{r+1} consists of two B_r trees, the root of one being a child of the root of the other. See Figure 1(a). In all figures, a triangle labeled \( Y \) represents the binomial tree B_r. Figure 1(b) shows an equivalent description of B_{r+1}: For any k, 0 ≤ k ≤ r, B_{r+1} consists of a B_r tree with additional children of the root that are themselves roots of B_k, B_{k+1}, . . . , B_r trees. For any node \( x \) in a binomial tree, \( \text{rank}(x) \) is the index \( r \) of the maximal subtree \( B_r \), rooted at \( x \). A binomial tree is an ordered tree, with the children of a node ordered by increasing rank. The last child of a node is the child of highest rank. Note that B_r has \( 2^r \) nodes and height \( r \).

In a tree data structure, each node \( c \) stores one item (including its key, denoted \( k(c) \)). Node \( c \) is good if it is the root, or if its parent \( p \) satisfies \( k(p) ≤ k(c) \); otherwise \( k(p) > k(c) \) \( c \) is bad. In a heap-ordered tree, all nodes are good. A binomial queue for \( 2^r \) items is a heap-ordered tree \( B_r \). A binomial queue for \( n \) items, \( n \) arbitrary, consists of at most \( \log n + 1 \) heap-ordered binomial trees, a tree corresponding to each one bit in the binary expansion of \( n \). The link operation for binomial queues takes two root nodes of equal rank \( r \) and creates a heap-ordered tree \( B_{r+1} \) by making the node with larger key a child of the smaller.

It seems difficult to process decrease_key in \( O(1) \) time and maintain heap order. The philosophy of relaxed heaps is to avoid this difficulty by permitting violations of heap order (whence the name). The rank relaxed heaps of Section 2 allow just one bad child per rank. The run relaxed heaps of Section 3 are less stringent and allow runs of bad children.

2. RANK RELAXED HEAPS

A relaxed tree is a tree data structure where some nodes are distinguished as active and any bad node is active. The terms relaxed binomial tree and relaxed binomial queue are interpreted according to this definition. A rank relaxed heap is a relaxed binomial queue that satisfies two conditions:

(a) For any \( r \) there is at most one active node of rank \( r \).

(b) Any active node is a last child.

Condition (a) implies there are at most \( \log n \) active nodes. Condition (b) is not crucial. It determines various programming details; we shall return to this point.

In the rest of this section, “relaxed heap” means “rank relaxed heap.”

The crux of the data structure is processing \( m \) decrease_key operations in time \( O(m) \). The next paragraph
gives a plan for accomplishing this. We then fill in the details of the plan and give the algorithm for delete-min.

The decrease_key algorithm rearranges nodes to keep the heap relaxed. It does this with three transformations. More precisely decrease_key(x; v) resets k(x) to v, after which it may stop or execute a transformation; a transformation does O(1) work, after which it may stop or execute another transformation. This gives rise to a sequence of transformations. To achieve the time bound, let Y denote the number of active nodes at any point in the algorithm. Initially there are no active nodes, so Y is zero. If resetting k(x) in decrease_key makes x bad, x is designated active. This increases Y by one. Each transformation either

(i) decreases α, or
(ii) does not change α and does not execute another transformation.

Observe that (i) and (ii) imply that any sequence of m decrease_keys uses time O(m): There are at most m type (ii) transformations since each is last in its sequence. There are also at most m type (i) transformations, since α is nonnegative and only decrease_key increases α. This argument will not be affected by delete-min operations (see the proof of Theorem 1).

To fill in the details of the plan we describe the transformations, at first informally. They are illustrated in Figures 2–5. In these figures an edge joining a child c to its parent p is labeled in one of four ways: An arrow from c to p indicates that c is good; a cross mark on the edge indicates c is active (and so can be good or bad); no mark indicates the status of c is unknown; an arrow pointing to the edge indicates c may be a new active node. The transformations use an operation “make node x the rank r child of node y.” This means that the entire B_r tree rooted at x becomes the B_r subtree of y. For each transformation, assume decrease_key (or a transformation) has created an active node α of rank r with parent p and grandparent g (α is actually a bad node, although this fact is not used).

The main idea is embodied in the pair transformation shown in Figure 2. It applies when α is the last child of p, and further, the relaxed heap already contains an active node a' of rank r, with parent p' and grandparent g'. Note that by condition (b), node a' is the last child of node p'. The transform removes the active nodes from their parents, so nodes a, a', p, p' all have rank r. Without loss of generality assume k(p) ≤ k(p'). The transform makes p' the rank r child of p (hence p remains a rank r + 1 node). Then it links α and a' to form a B_r tree with root c (so c is a or a'). It makes c the rank r + 1 child of g'. If c is now a bad child it is active, and a transformation is done for it. Note that c need not be the last child of g'.

The last detail concerns linking α and a'. In general, as in Figure 3, suppose nodes q and q' of rank s + 1 have just been linked, making q the new root. If x, the rank s child of q, is active, it now violates condition (b) of the relaxed heap structure. Figure 3 compensates for this by performing a cleaning operation. It uses the fact that x', the rank s child of q', is good if x is active. This follows from the definition of a relaxed heap. (Care should be taken here, since the transformations are applied to heaps where the relaxed heap structure has been violated; however, we will use the clean operation only when this deduction is valid.) The operation repairs the damage by interchanging x and x'. In the following, to clean node q means to apply the operation of Figure 3 if x is active; otherwise do nothing. To combine two nodes means to link them and then clean the new root. Thus, in Figure 2, the pair transformation combines α and a'.
Research Contributions

$a'$ is a last child but $a$ is not. This is the case in the good sibling transformation described below, which does a pair transform of this kind.

The remaining transformations are "sibling transformations." For these assume that $a$ has rank $r + 1$ sibling $s$. The active sibling transformation applies when $s$ is active, as shown in Figure 4. The definition of a relaxed heap implies that $s$ is a last child, so $\text{rank}(p) = r + 2$.

The transform removes the two active children from their parent $p$. It combines $p$ and $a$, making $a$ a rank $r + 1$ node. Then it combines $a$ and $s$ into a tree whose root $c$ becomes the rank $r + 2$ child of $g$. If $c$ is now bad, a transformation is done for it. The active sibling transformation achieves property (i), since two active nodes are replaced by at most one.

The good sibling transformation applies when $s$ is good. Let $c$ be the last child of $s$; hence $\text{rank}(c) = r$. There are two cases. If $c$ is active the algorithm does a pair transform for $a$ and $c$ (as depicted in Figure 5). As noted previously, the transformation works correctly even though $a$ is not a last child; the only delicate point is to make sure that if $k(p) = k(s)$ the algorithm makes $s$ the child of $p$, not vice versa. This case achieves property (i), as above. The second case occurs if $c$ is a good child. In this situation the cleaning operation of Figure 3 is applicable. The transform cleans $p$, making $a$ a bad last child of $s$. It then processes $a$ as a new active last child: If there is an active node of rank $r$ a pair transform achieves property (i); otherwise the sequence of transformations stops, achieving (ii).
Now we describe the delete\-min operation. Note that in most applications (e.g., all those in [3, 5]) it is unnecessary to reclaim the storage used by a deleted node. We give two implementations of delete\-min, the simpler of which does not reclaim storage. Both implementations start by finding the smallest node x. Since x is either active or the root of a tree in the queue, it can be found in O(log n) time.

The nonreclaiming algorithm sets k(x) to \( \infty \) and changes rank(x) from r to 0. Then it merges x and its former children into a new rank r node, by repeatedly combining the two nodes of smallest previous rank. The new rank r node replaces x in the tree. Note that x becomes a leaf and will not participate in any future transformation.

The reclaiming algorithm is similar. It deletes x and removes the root node of smallest rank y from its tree; this makes the previous children of y into roots with the smallest ranks in the queue. Then it processes y (not x) and the former children of x as in the nonreclaiming algorithm.

Figure 6 gives a more detailed description of the algorithm in pseudo-Algo. The delete\-min implementation uses the reclaiming algorithm. The following data structure is assumed. Each node x has a record containing k(x), rank(x), and pointers to its last child, its two neighboring siblings, and its parent. (A sibling pointer is needed in the sibling transformations. Hence pointers to both siblings are needed to allow nodes to be moved. A last child pointer is needed in the good sibling transformation.) There is a dummy node \( \rho \) treated as the root of the entire queue: each root of a binomial tree has parent \( \rho \), so the roots can be treated as siblings and are not special cases. Further, \( k(\rho) = \infty \) and \( \rho \) is its own parent. In addition, there is an array \( A[0 \ldots \log n - 1] \); each \( A(r) \) is a pointer to the active child of rank r, if it exists. The A array is used to check if a node is active, e.g., promote tests if s is active by checking the condition \( A(r + 1) = s \). It should be clear that this data structure supports the desired operations and can be maintained in time \( O(1) \) per transformation.

It should be noted that the data structure can be initialized in \( O(n) \) time (assuming, as is often the case,

that the number of items \( n \) is known in advance). One way is to construct a binomial queue on \( n \) items, with each key equal to \( \infty \). The operation insert(x) is done by executing decrease\_key(x, k(x)).

**Theorem 1.** Rank relaxed heaps correctly process a sequence of \( m \) decrease\_key and \( k \leq n \) delete\_min operations in time \( O(m + k \log n) \).

**Proof.** It is easy to check that the algorithm maintains the following invariant: At the start of each call to promote, making the edge between a and its parent good gives a valid relaxed heap structure. This implies correctness. (Note that an active node can be good or bad: An entry in the A array starts out as a bad child; it may become good without being processed in a transformation, if the key of its parent is sufficiently decreased.)

For the timing, observe the decrease\_key routine uses \( O(1) \) time and the delete\_min routine uses \( O(\log n) \) time. The time for transformations is bounded by the argument given as the plan for the algorithm (delete\_min decreases \( \alpha \) by one or zero, and so only improves the bound).

The algorithm of Section 4 uses several other properties of relaxed heaps. We start with an extension of the theorem.

**Corollary 1.** Any subsequence of \( k \) consecutive decrease\_key operations is processed in time \( O(k + \log n) \).

**Proof.** Let the subsequence start with \( \alpha_0 \) active nodes and end with \( \alpha_k \) active nodes. Each decrease\_key increases \( \alpha \) by at most one. Hence the number of type (i) transformations (that decrease \( \alpha \)) is at most \( \alpha_0 + k - \alpha_k \). The number of type (ii) transformations (that are last in their sequence) is at most \( k \). Since \( \alpha_0, \alpha_k \leq \log n \), the number of transformations, and hence the total time, is \( O(k + \log n) \).

As noted in the proof of the theorem, a single delete\_min operation in a sequence uses time \( O(\log n) \). Section 4 uses two other priority queue operations: The operation find\_min (defined in Section 4) is implemented exactly like the first part of delete\_min. Hence it uses time \( O(\log n) \). The operation delete(x) can be implemented by...
Research Contributions

decreasing \(k(x)\) to \(-\infty\) and then executing delete_min. Hence it uses time \(O(\log n)\).

The transformations here were selected for economy of description. Different ones would probably be more efficient in practice. For instance, in the combine routine, a more productive way to clean the tree of the new root is to repeatedly make an active bad child of the root into the new root, until the root has no active child. This approach has the advantage of decreasing the number of active nodes. Alternatively, cleaning can be eliminated entirely by dropping the requirement that an active node in a relaxed heap be last. Instead, the active sibling transformation is extended to process a string of consecutive active siblings: the processing is similar to delete_min. Such transformations might work better if the processing of a decrease_key operation is delayed until the next delete_min (in the hope of getting longer strings of active nodes).

A drawback in practice is the large number of pointers per node. The sibling and last child pointers are redundant, and increase both time and space. Those three pointers per node can be replaced as follows. Each node \(x\) has a child table, i.e., an array of pointers to its children, indexed by rank. The siblings of a node can be read from the child table of its parent in \(O(1)\) time.

```
procedure decrease_key(x, v)
begin k(x) = v; promote(x) end;

procedure promote(x)
begin
if x is the last child of v then begin
  if A(r) = nil then A(r) = v
  else if A(r) is then pair_transform end
else if v is active then active_sibling_transform
else good_sibling_transform
end;

procedure combine(a, b, c)
begin
if a and b have equal rank k then
  link a and b to a rank k + 1 tree with root c; clean c;
end;

procedure pair_transform
begin
a' = A(r); A(r) = nil; let a' have parent p and grandparent g;
remove a and a' from their parents
\(a, a', p, p'\) now have rank r + 1;
\(k(p) = k(p')\);
make p' the rank r child of p;
combine(a, a', c); make c the rank r + 1 child of g;
if A(r + 1) = p' then A(r + 1) = c else promote(c);
end;

procedure active_sibling_transform
begin
remove a, s, p from their parents
\(a, s, p\) now have rank r, and s has rank r + 1;
A(r + 1) = nil;
combine(p, a, s);
\(combine(a, s, c); make c the rank r + 2 child of g;
if A(r + 2) = p then A(r + 2) = c else promote(c);
end;

procedure good_sibling_transform
begin
let c be the last child of s;
if c is active then pair_transform
else begin clean p; if p is good, is a bad last child,
  promote(p) and end;

procedure delete_min(x)
begin
x = the node of smallest key from among the root nodes of the queue and the A(r) values;
y = the root node of smallest rank:
remove y from its tree, creating rank(y) new roots with the smallest ranks in the queue;
if y had an active last child it remains active, with no harm done;
if \(x = y\) then begin
  for each child \(c\) of \(x\) in order of increasing rank
    do combine(y, c, y);
delete x from its tree and replace it by y;
  let p be the parent of y;
  if A(rank(y)) = x then A(rank(y)) = (if k(y) < k(p) then y else nil);
end end;
```

The disadvantage of child tables is that a rank \(r\) node needs a table of \(r\) entries, and \(r\) can be \(\log n\). Hence a uniform node size increases the space requirement to \(\Theta(n \log n)\). However, a two-tiered data structure can reduce the space to linear—in fact to \((1 + \epsilon)n\) words, for any \(\epsilon > 0\). Further, the disadvantage of a nonuniform data structure seems to be compensated by the simplicity of the second tier. Details are as follows.

Form \(n/\log n\) groups of \(\log n\) items (the last group may have fewer items). The grouping can be arbitrary. For instance, if the items are numbered from 1 to \(n\) and accessed by number (as can be arranged in many applications) the groups are consecutively numbered items; no extra storage is needed for the group structure. Each group is represented by a node \(g\) in a relaxed heap of \(n/\log n\) nodes; the key of \(g\) is the smallest key of an item \(x\) in the group corresponding to \(g\); \(g\) stores a pointer to \(x\) (so \(k(g)\) need not be stored); the relaxed heap uses child tables. To do decrease_key(x, v) the algorithm updates the key of the group corresponding to \(x\); if it decreases, the group is promoted in the relaxed heap. To do delete_min(x), the group corresponding to \(x\) is scanned for the undeleted item \(y\) with smallest key: a pointer to \(y\) replaces the pointer to \(x\) in the relaxed heap node, and the algorithm follows the delete_min algorithm without reclamation. Clearly the storage is

![FIGURE 6. Rank Relaxed Heap Algorithm](image-url)
O(n); if the relaxed heap uses w words per node, choosing a group size of \( w/e \) gives \( \alpha n \) storage for the relaxed heap. The time bound is unchanged if \( w = O(\log n) \).

Child tables may be used with Fibonacci heaps but are not as effective. In Fibonacci heaps an arbitrary child of a node may be deleted without replacement. This causes arbitrary deletions of entries in the child tables. This forces the algorithm to keep track of the free entries in a child table, by a free storage list. This overhead is not present in relaxed heaps, where only a highest rank entry gets deleted without replacement.

The algorithm given so far can be used instead of Fibonacci heaps in two important applications, Dijkstra's shortest path algorithm and the computation of minimum spanning trees. It is easy to extend the algorithm to support all the operations supported by Fibonacci heaps. A node can be inserted in \( O(1) \) amortized time. This is done by the technique given above for initializing the heap, if \( w \) is known in advance. Alternatively, the algorithm makes the inserted node into a B\( _0 \) tree, and then repeatedly links B\( _1 \) trees of equal rank until this is no longer possible. The amortized time for an insert is \( O(1) \), since in a sequence of \( r \) inserts and \( d \) deletes the number of times \( b \) trees are linked by inserts is at most \( i + d \log n \).

To merge two relaxed heaps, repeatedly link \( b \) trees of equal rank, until this is no longer possible. Use the \( A \) array of the first heap as the \( A \) array of the merged runs can be used to handle this situation. The details of the transformations.

The algorithm keeps track of the run- and singleton-structure of the active nodes, which we refer to as the run-singleton structure. The run-singleton structure can change when a transformation creates a new bad node or rearranges nodes. The bookkeeping details for the run-singleton structure are relatively straightforward and will be postponed until after the transformations.

There are two transformations, one for runs and one for pairs of singletons. In this section a transformation does \( O(1) \) work and decreases \( \alpha \), and then stops. Unlike Section 2, there are no sequences of transformations. Instead, if a transformation creates a new bad node \( b \), \( b \) is added to the run-singleton structure. Transformations do the rearrangements of Figures 2-5, but these figures are interpreted with one difference: Nodes are linked rather than combined. There is no need to combine nodes since the new definition of relaxed heap does not require a cleaning operation. Now we describe the details of the transformations.

The singleton transformation is a combination of transformations of Section 2. It is given two singletons \( a, a' \) of equal rank \( r \). We use the same notation as Section 2—\( a \) and \( a' \) have parents \( p \) and \( p' \) and grandparents \( g \) and \( g' \), respectively. There are three cases. The first is when \( a \) and \( a' \) are both last children. Then Figure 2 applies: Without loss of generality assume \( k(p) \leq k(p') \).

The transform makes \( p' \) the rank \( r \) child of \( p \), and links \( a \) and \( a' \) to form a \( R_1 \) tree. The only difference is how the transform affects the run-singleton structure: When \( a \) and \( a' \) are singletons the only change is a possible new run or singleton if the new child of \( g' \) is bad. When \( a \) or \( a' \) starts out in a run that run also changes.

The second case of the singleton transform is when exactly one of \( a \) and \( a' \), say \( a \), is not a last child. Thus \( a \) has a rank \( r + 1 \) sibling \( s \); since \( a \) is a singleton, \( s \) is good. Let \( c \) be the last child of \( s \). If \( c \) is bad Figure 5 applies: a first-case singleton transform is done for \( a \) and \( c \) (note that \( c \) may be in a run). If \( c \) is good Figure 3 applies: a cleaning operation makes \( a \) a last child.

The second transformation is the run transformation. Let \( a \) be the largest rank child of the given run. Let \( a \) have rank \( r \), parent \( p \) and rank \( r - 1 \) sibling \( t \); \( t \) is active, since it is in a run. The first case is if \( a \) is a last child. Then the transformation of Figure 4 applies:
and $p$ are linked, and then $t$ and $u$ are linked. This decreases $a$ as desired. (The only possible change in the run-singleton structure involves the new child of $g$.)

The last case is when $a$ is not last (see Figure 7). Then $a$ has a rank $r + 1$ sibling $s$, which is good (since it is not in the run). Let $d$ and $c$ be the rank $r - 1$ and rank $r$ children of $s$, respectively. Children $t$, $a$, $s$, $d$, and $c$ are removed from their parents. Nodes $d$ and $c$ are made the rank $r - 1$ and rank $r$ children of $p$, respectively. Since $s$ was good, this does not increase $a$. Then $t$ and $s$ are linked to form a $B$, tree with root $t$; next $t$ and $u$ are linked, and the new root is made the rank $r + 1$ child of $p$. The result is that $a$ decreases by one, so this transformation operates as desired. (The possible changes in the run-singleton structure involve the new children of $p$ of ranks $r - 1$ through $r + 1$, and the remainder of a possible run initially containing $d$.) This completes the description of the run transformation.

Now we give the details of the run-singleton data structure. A bit is used to mark each active node. Each rank $r$ has a list $S(r)$ of all rank $r$ singletons. The pair list is a list of ranks $r$ that have $|S(r)| \geq 2$. The run list is a list containing the largest rank node of each run. All lists are doubly-linked, and each node points to its occurrence in a list (if any). In addition, the nodes of the binomial trees have the same pointers as in Section 2.

Now we discuss updating the run-singleton data structure. We use a subroutine $\text{add-singleton}(x)$ that adds a new singleton $x$ to the data structure. Specifically for $r = \text{rank}(x)$, it adds $x$ to list $S(r)$. If this makes $|S(r)| = 2$ then $r$ is added to the pair list. Similarly, the subroutine $\text{remove-singleton}(x)$ removes $x$, a node that was previously a singleton, from the data structure. For $r = \text{rank}(x)$ it deletes $x$ from $S(r)$. If this makes $|S(r)| = 1$ then $r$ is deleted from the pair list.

The following procedure updates the run-singleton

**FIGURE 7. Run Transformation**
data structure when a node $a$ becomes active. Let $a$ have rank $r$, with rank $r+1$ siblings $s$ and rank $r-1$ sibling $t$ ($s$ or $t$ may not exist). Node $a$ is marked active. Then the appropriate one of these four cases is executed:

1. If neither $s$ nor $t$ is active then $a$ is a singleton. Do add_singleton($a$).
2. If $s$ is active but $t$ is not, then $a$ is in a run. If $s$ was previously a singleton, do remove_singleton($s$) and add $s$ to the run list.
3. If $s$ is not active but $t$ is, then $a$ is the largest node of a run. If $t$ was previously a singleton, do remove_singleton($t$); otherwise ($t$ was previously on the run list) delete $t$ from the run list. Add $a$ to the run list.
4. If $s$ and $t$ are active, then $a$ is in a run. Process $s$ exactly as in case (2); process $t$ exactly as in case (3). (Do not add $a$ to the run list.)

This concludes the update algorithm.

To do decrease_key($x$, $v$), $k(x)$ is changed to $v$. If this makes $x$ bad it is inserted into the run-singleton structure, using the update algorithm described. The rest of the processing ensures $\alpha \leq \log n + 1$. If the pair list is nonempty, a rank $r$ is removed from it; a singleton transformation is done for the first two elements of $S(r)$ (any necessary changes to the run-singleton structure are made); if we still have $|S(r)| \geq 2$, $r$ is added to the pair list. Similarly, if the run list is nonempty a node $a$ is removed from it; a transformation is done for the run ending at $a$ (any necessary changes to the run-singleton structure are made); if $a$ is still left in the run previously containing $a$, the largest node is added to the run list or a singleton list, as appropriate.

The time for decrease_key($x$, $v$) is clearly $O(1)$. Correctness follows from the fact that if $x$ becomes active and makes $\alpha > \log n + 1$ then two active nodes have the same rank. Hence the pair list or run list is nonempty, so a transformation decreases $\alpha$ to a permissible value.

The delete_min routine is similar to Section 2. First the minimum node $x$ is found. This involves examining all tree roots and all active nodes. To find the active nodes, singletons are found using the 5 lists. To find a run, its largest rank node is found on the run list; the remaining nodes are found by following sibling pointers until an unmarked node is reached. The minimum node $x$ is replaced by a node resulting from linking all of its former children. (This may be done reclaiming storage or not, as in Section 2.) Various nodes that become good are removed from the run-singleton structure. (The procedure for removing nodes from the structure is similar to the above update routine.) The definition of relaxed heap ensures that the total time is $O(\log n)$.

The data structure can be extended to handle the operation insert($x$) in $O(1)$ time. The approach is similar to decrease_key. Let $\tau$ denote the number of trees in any collection of trees. A run relaxed heap is now defined as a collection of relaxed binomial trees with $\alpha \leq \log n + 1$ and $\tau \leq \log n + 1$ (note that $\alpha$ refers to the number of active nodes in the entire collection of trees). The algorithm keeps a pointer $T(r)$ to a binomial tree of rank $r$, for each $r$; the remaining trees are kept in the tree list, a list of pairs of trees of the same rank.

The subroutine add_tree($x$) puts a new binomial tree with root $x$ into the structure: Letting $r = \text{rank}(x)$, if $T(r) = \text{nil}$ then $x$ becomes $T(r)$; otherwise $x$ and $T(r)$ are made into a pair which is added to the tree list and $T(r)$ becomes $\text{nil}$. The operation insert($x$) makes $x$ into a $B_0$ tree and does add_tree($x$). Then it decreases the number of trees, if possible: It removes the first pair from the tree list, links the two trees to form a tree with root $y$, and does add_tree($y$). The time for insert($x$) is clearly $O(1)$.

Correctness follows from the fact that if in insert($x$), add_tree($x$) makes $\tau > \log n + 1$ then the tree list is nonempty. Hence the second step of insert decreases $\tau$ to a permissible value.

The new data structures have no effect on decrease_key, since the transformations and the run-singleton data structure are independent of the arrangement of the trees. The reclaiming version of delete_min creates new trees; it is modified to repeatedly link trees of equal rank, so the tree list becomes empty. This does not change the time bound since $O(\log n)$ trees are involved.

**Theorem 2.** Run relaxed heaps correctly process decrease_key and insert in $O(1)$ time and delete_min in $O(\log n)$ time.

Many details of the run relaxed heap algorithms can be modified without changing the above bounds. For instance the pair transform can be simplified if it is only done when there are no runs.

The next section uses two other priority queue operations, find_min and delete(x). These are implemented similarly to their implementation in rank relaxed heaps. Each such operation uses time $O(\log n)$.

### 4. Parallel Network Algorithms

This section uses relaxed heaps to give an efficient parallel implementation of Dijkstra's shortest path algorithm. It begins by reviewing some ideas from parallel computation and summarizing Dijkstra's algorithm. Then it gives the parallel shortest path algorithm and applications to other network problems.

A Parallel Random Access Machine (PRAM) consists of $p$ synchronized processors accessing a common memory. On an EREW (Exclusive Read Exclusive Write) PRAM, at most one processor can access a given memory cell in any instruction cycle. It is convenient to index the processors as $p_i$, $1 \leq i \leq p$.

Recall two fundamental operations for processor communication on a PRAM: To broadcast a value $v$ means that each processor reads a copy of the value $v$. This can be accomplished using an array $A[1 \ldots p]$, where initially $v$ is stored in $A[1]$; the value $v$ is copied into all other entries of $A$, and then in parallel processor $p_i$, reads $A[i]$. A broadcast operation can be done in time $O(\log p)$ on an EREW PRAM.
Research Contributions

The second operation is to compute the minimum, i.e., given an array of values \( A[1 \ldots p] \), set \( A[i] \) to \( \text{min}(A[i]) \) for \( 1 \leq i \leq p \). This can also be done in time \( O(\log p) \).

Next recall the shortest path problem. (More detailed discussions can be found in [10, 13].) Given is a directed graph \( G \) with a distinguished vertex \( s \), where each edge \( e \) has a real-valued length \( l(e) \). Let \( V \) denote the set of vertices, and let \( n \) and \( m \) denote the number of vertices and edges, respectively. The problem is to find, for each vertex \( v \), the distance from \( s \) to \( v \), i.e., the length of a shortest path from \( s \) to \( v \). Other useful information, such as the shortest paths themselves, can be computed by simple modifications of the algorithm we describe, without changing any resource bounds.

We have omitted such modifications for simplicity. Assume that every vertex can be reached by a path from \( s \). This assumption is also made for simplicity and can be eliminated without changing any resource bounds. The last assumption is that all edge lengths are nonnegative. This assumption is crucial. It allows the problem to be solved using Dijkstra's algorithm, a variant of breadth-first search. The following paragraphs summarize this algorithm.

The algorithm uses two data structures. It maintains \( S \) as the set of vertices whose distance is known. It also stores for each vertex \( v \) a value \( d(v) \). If \( v \in S \) then \( d(v) \) equals the desired distance from \( s \) to \( v \). If \( v \notin S \) then \( d(v) \) equals the length of a shortest path from \( s \) to \( v \) whose vertices are in \( S \cup \{v\} \); equivalently, \( d(v) = \min\{d(u) + l(uv) \mid u \in S\} \).

The algorithm starts by initializing \( d(s) \) to zero, all other values \( d(v) \) to \( \infty \), and \( S \) to the empty set. Then it repeatedly performs an iteration that adds one vertex to \( S \), until \( S \) contains all \( n \) vertices. The iteration performs the following three steps:

Step 1. Set \( v \) to a vertex not in \( S \) such that \( d(v) \) is minimum.

Step 2. Add \( v \) to \( S \).

Step 3. For every edge \( vw \) from \( v \) to a vertex \( w \in S \), set \( d(w) = \min\{d(w), d(v) + l(vw)\} \).

This algorithm can be implemented efficiently on a sequential machine by maintaining the set of vertices \( V-S \) as a priority queue, where the key of a vertex \( v \) is \( d(v) \). Steps 1-2 amount to a \textit{delete-min} operation. Step 3 is done by examining each edge \( vw \) directed from \( v \), and if the value \( d(w) \) should be decreased, doing the operation \( \text{decrease-key}(w, d(v) + l(vw)) \). Using a relaxed heap (or Fibonacci heap) for the priority queue gives total time \( O(m + n \log n) \) for priority queue operations. If the graph is represented by an appropriate representation (such as a collection of adjacency lists of edges directed from each vertex) the total time is dominated by the priority queue operations. This gives time \( O(m + n \log n) \) and space \( O(m) \) for the entire algorithm.

Now we present a parallel implementation of the algorithm. The idea is to take advantage of the obvious parallelism in Step 3, where all edges from \( v \) can be processed at the same time. First we adapt the data structure of the sequential algorithm to accomplish this. The representation of the graph is split among the processors. More precisely, each processor has a collection of adjacency lists, one for each vertex. The edges directed from a given vertex \( v \) are divided evenly among the processors, so that if there are \( m \) such edges, each processor has \( \lfloor m/n \rfloor \) or \( \lceil m/n \rceil \) edges. (This representation can be easily constructed or simulated—we shall return to this point.)

Each processor \( p \) maintains its own priority queue. The queue records information about the edges that \( p \) scans, similar to the sequential algorithm. More precisely at the start of each iteration, the priority queue for \( p \) contains each vertex \( v \in V-S \) with key \( d(v) = \min\{d(u) + l(uv) \mid u \in S, uw \text{ is an edge in the adjacency structure of } p \} \). The priority queue is implemented as a relaxed heap of either type, random or rank relaxed.

Now we describe the parallel algorithm. We begin with a description of an iteration of Dijkstra's algorithm. To do Step 1, each processor does \textit{find-min} on its priority queue. Then the smallest of all the priority queue minima is found by doing a parallel minimum computation. Let \( v \) be the vertex found to have overall minimum key \( d(v) \). To do Step 2, the values \( v \) and \( d(v) \) are broadcast to each processor. Then each processor does \textit{delete}(v) on its priority queue. Finally to do Step 3, each processor \( p \) scans the edges from \( v \) in its adjacency structure. For each such edge \( vw \) if the value \( d(w) \) should be decreased, \( p \) does the operation \( \text{decrease-key}(w, d(v) + l(vw)) \) in its priority queue.

It is important that the processors remain synchronized in this algorithm. More precisely, each processor must start the execution of each Step of each iteration at the same time. To achieve this, each processor spends exactly \( t \) time units in a Step, where \( t \) is the exact worst-case time bound for the Step. If a processor finishes its work early, it counts out time units until \( t \) is reached. (For example, this may occur in Step 3, if a processor does not do a transformation for every \textit{decrease-key} operation.)

To start the parallel algorithm, each processor initializes its priority queue as in the sequential algorithm. Then the first iteration is begun.

Observe that at the end of the algorithm each processor has recorded the correct distance \( d(v) \) for each vertex \( v \).

Now we estimate the resource requirements. For the space observe that in addition to the edges of the graph, each processor uses \( O(n) \) space, for its adjacency structure, distances, and priority queue. Since we will assume \( \rho \leq m/n \) this extra space is \( O(m) \).

Now we show that the time is \( O(m/n + n \log n) \). First consider the operations not involving priority queues. The processor communication time for \( n \) minimum computations in Step 1 and \( n \) broadcasts in Step 2 is \( O(n \log p) \). To examine edges on adjacency lists in Step 3, note that in the entire algorithm \( m \) edges are scanned in Step 3. There are at most \( m/p \) time units where every processor scans an edge. In addition there
are at most \( n \) time units where some but not all processors scan an edge. This gives total time \( O(m/p + n) \).

Now consider the time for priority queue operations. In \( n \) executions of Step 1 each processor does a \texttt{find\_min} operation. In \( n \) executions of Step 2 each processor does a \texttt{delete} operation. These operations use time \( O(n \log n) \). Finally in \( n \) executions of Step 3 each processor does a total of at most \( m/p + n \) \texttt{decrease\_key} operations. If run relaxed heaps are used this takes time \( O(m/p + n) \). If rank relaxed heaps are used this takes time \( O(m + n \log n) \), by Corollary 1. (It may be helpful to elaborate here on the implementation of Step 3 using rank relaxed heaps. Suppose vertex \( v \) has \( k \) edges directed from it. Then the worst-case time bound for Step 3 is \( O(\alpha(a/p) + n \log n) \). If a processor finishes before these many time units have elapsed, it waits to begin the next execution of Step 1.)

This proves the main result of this section.

**THEOREM 3.** The shortest path problem on a directed graph with nonnegative edge lengths can be solved in time \( O(m/p) \) and space \( O(m) \), on an EREW PRAM with \( p \leq m/[n \log n] \).

This result is an optimal algorithm for the shortest path problem for the specified processor range. Although the result can be achieved using either variant of relaxed heaps, it is not clear whether the result can be achieved with Fibonacci heaps. The reason is that the processors must remain synchronized, in particular after each execution of Step 3. Using Fibonacci heaps it is possible that in each iteration a (different) processor lags far behind the others, and they must wait. This leads to idle time and less than optimum speedup.

We should also comment on the representation of the graph for the parallel algorithm. It is easy to derive the desired representation from any reasonable representation of the graph. Suppose the graph is given as sequentially-stored adjacency lists. Specifically, each vertex \( v \) has a list \( A(v) \) of edges directed from \( v \), stored in consecutive locations; in addition, \( v \) has pointers \( \text{first}(v) \) and \( \text{last}(v) \) to the first and last edges of \( A(v) \), respectively. The data structure for the parallel algorithm is easily constructed. Alternatively it may be more efficient to avoid constructing the data structure, and simply simulate it. That is, in Step 3 to scan the edges from \( v \), assume that each processor knows \( \text{first}(v) \) and \( \text{last}(v) \), from the broadcast of Step 2. Each processor computes \( a = |A(v)| = \text{last}(v) - \text{first}(v) + 1 \), and scans \( a/p + 1 \) or \( a/p \) edges. One way to do this is to have \( p \) processors scan the \( j \)th edge from \( v \) for \( j = i + \text{first}(v) + kp \), where \( k \) ranges over all nonnegative integers that give \( j \leq \text{last}(v) \).

The shortest path problem can be solved to solve many other network problems. We note several such applications. (Complete definitions of these problems can be found in [10, 13].)

**COROLLARY 2.** Consider a graph with \( n \) vertices and \( m \) edges, \( m \geq n \log n \), and a EREW PRAM with \( p \leq m/[n \log n] \).

(a) A minimum spanning tree can be found in time \( O(m/p) \).

(b) The assignment problem can be solved in time \( O(nm/p) \). The all-pairs shortest path problem on a directed graph with possibly negative edge lengths can be solved in the same bound.

(c) A minimum cost flow can be found in time \( O(m^2 \log n)/p \). The uncapacitated transportation problem can be solved in time \( O(nm \log n)/p \).

In all algorithms the space is \( O(m) \).

Proof. We sketch the applications, leaving the details to the interested reader.

(a) Dijkstra's algorithm to find a minimum spanning tree (also due to Prim, see [13]) is essentially the above shortest path algorithm, with the assignment of Step 3 changed to \( d(w) \leftarrow \min(d(w), I(vw)) \). Hence the parallel algorithm is virtually unchanged.

(b) The Hungarian algorithm for the assignment problem [10] consists essentially of \( n \) applications of Dijkstra's algorithm. [4] shows that the all-pairs problem can be solved by solving an assignment problem followed by \( n \) applications of Dijkstra's algorithm.

(c) Orlin's algorithm [11] reduces the minimum cost flow problem to \( O(m \log n)/p \) shortest path problems. The shortest path problems can be solved by Dijkstra's algorithm, provided certain length zero edges are chosen preferentially. It is easy to incorporate this modification into our parallel algorithm. Other overhead in the algorithm is also easy to parallelize. Similar remarks apply to the transportation problem.

We close by comparing these results with previous work. Previous parallel minimum spanning tree algorithms are based on the parallelism inherent in Boruvka's algorithm [13]. The algorithms achieve optimal speedup of Boruvka's algorithm, i.e., time \( O(\log n) \) [9] or slightly more for more processors than we allow here (close to \( m \) processors) on an EREW PRAM [9] or more permissive models [1, 8]. In contrast, our algorithm achieves the optimal time bound \( O(m/p) \), but allows fewer processors.

Our bound for the assignment problem is strongly polynomial. Several algorithms achieve greater parallelism, although they are based on cost-scaling and so are only weakly polynomial. Specifically [6] achieves time \( O(n^2 \log N \log(2p)/p) \) when the costs are integers of magnitude at most \( N \), for \( p \leq m/(\sqrt{n} \log n) \) on an EREW PRAM. This bound does not apply to the all-pairs shortest path problem.

For the minimum cost flow problem [7] achieves time \( O(n^2 \log N \max[\log(nN)/m \log n]) \) using \( n \) processors and \( O(n^2) \) space. To compare this with (c) assume the maximum number of processors, \( p = \max[m/(\sqrt{n} \log n), 1] \). Their strong polynomial bound uses more time, processors and space than (c). On the transportation problem our time bound \( O(n^2 \log^2 n) \) is similar to their weakly polynomial bound, but uses fewer processors and less time.
space. However on the flow problem their weakly polynomial bound is superior.

Acknowledgments. Ruth Shrairman would like to thank Professor Rafael Hassin of Tel Aviv University for his support and helpful comments.

REFERENCES

CR Categories and Subject Descriptors: F.2.2 [Data Structures]: F.2.2 [Nonnumerical Algorithms and Problems]: Computations on discrete structures; G.2.1 [Combinatorics]: Combinatorial Algorithms; G.2.2 [Graph Theory]; G.3.1 [Additional Key Words and Phrases]: Amortization, parallel computation, priority queue, shortest path, worst-case bound

ABOUT THE AUTHORS:

JAMES R. DRISCOLL is an assistant professor of mathematics and computer science at Dartmouth College. His research interests are centered on the design and analysis of data structures and algorithms. Author's present address: James R. Driscoll, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755; james.r.driscoll@dartmouth.edu.

HAROLD N. GABOW is a full professor in the computer science department of the University of Colorado at Boulder. His research interests are in the design and analysis of algorithms, especially for problems on graphs and matroids. He is currently working on efficient matroid algorithms for problems with practical applications such as rigidity of physical structures and controllability of dynamic systems. His network flow model of medical attending schedules is used to schedule the doctors at Denver General Hospital.

RUTH SHRAIRMAN is currently a Ph.D. candidate in the computer science department of the University of Colorado at Boulder. Her research interests include combinatorial optimization and data structures. She is a member of the Operations Research Society of Israel. Her electronic mail address is: shraima@boulder.col.edu. Authors' present addresses: Harold Gabow and Ruth Shrairman, Department of Computer Science, University of Colorado, Boulder, CO 80309.

ROBERT E. TARJAN is the James S. McDonnell Distinguished University Professor of Computer Science at Princeton University and a Distinguished Member of the technical staff at AT&T Bell Laboratories. He was the 1986 coreipient of the A.M. Turing Award. He is a member of the National Academy of Sciences and the National Academy of Engineering. His current research interests include design and analysis of algorithms, data structures, network optimization, and computational geometry. Author's present address: Robert E. Tarjan, Computer Science Department, Princeton University, Princeton, NJ 08544; also at 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.

In response to membership requests ... CURRICULA RECOMMENDATIONS FOR COMPUTING

Volume I: Curricula Recommendations for Computer Science
Volume II: Curricula Recommendations for Information Systems
Volume III: Curricula Recommendations for Related Computer Science Programs in Vocational-Technical Schools, Community and Junior Colleges and Health Computing

Information available from the ACM Order Dept., 1-800/342-6626 (in Maryland, Alaska or Canada, call (301) 528-4261).