Worst-Case Union-Find with Fast Deletions

Stephen Alstrup
Inge Li Gørtz
Theis Rauhe
Mikkel Thorup

IT University Technical Report Series

TR-2003-25

ISSN 1600–6100 **December 2003**

ISSN 1600–6100

ISBN 87-7949-034-4

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web `www.itu.dk`

Worst-Case Union-Find with Fast Deletions

Stephen Alstrup[1], Inge Li Gørtz[1], Theis Rauhe[1], and Mikkel Thorup[2]

[1] Theory Department, The IT University of Copenhagen, Denmark. Email: {stephen,inge,theis}@it-c.dk.
[2] AT&T Research Labs, USA. Email: mthorup@research.att.com.

Abstract. In the classical union-find problem we maintain a partition of a universe of elements into disjoint sets subject to the operations union and find. Kaplan et al. [SODA 2002] studied an extension of this problem, where also deletions are allowed. They gave a data structure that for any fixed k supports *find(x)* and *delete(x)* in $O(\log_k n)$ worst-case time and union in $O(k)$ worst-case time, where n is the number of elements in the set containing x. They asked if the delete time could be made faster than their find time. We answer this question affirmatively showing how to get the delete time down to $O(\log^* n)$, while keeping the same time bounds for union and find.

1 Introduction

In the classical union-find problem we maintain a partition of a fixed universe of N elements into disjoint sets subject to the operations union and find. The starting point is a collection of sets each containing a single element, and some of these sets are then subsequently combined, but the underlying universe remains the same. A classical union-find data structure allows the following operations on a collection of disjoint sets:

– *make-set(x)*: Creates a set containing the single element x, and returns the name of the set.
– *union(A,B)*: Combine the sets A and B into a new set, destroying the sets A and B, and return the name of the new set.
– *find(x)*: Finds and returns (the name of) the set that contains x.

Kaplan et al. [4] studied the *union-find with deletions* problem, where the universe is no longer fixed. That is, in addition to the above three operations they allow operations that insert or delete elements in the universe.

In the union-find with deletions problem we in addition to the above three operations allow an insert and a delete operation:

– *insert(x,A)*: Inserts an element x not yet in any set into set A.
– *delete(x)*: Deletes x from the set containing it.

Note that the delete operation does not get the set containing x as a parameter.

As noted by Kaplan et al. we can in a straight-forward way extend a classical union-find data structure to also support an *insert(x,A)*. This can be done by first perform $B =$ *make-set(x)* followed by *union(A,B)*.

1.1 Previous Work

For the classical union-find problem (without deletions) Smid [6] (building upon a previous result of Blum [2]) gave for any fixed k a data structure that supports union in worst-case $O(k)$ time and find in worst-case $O(\log_k n)$ time, where n is the size of the corresponding set. These time bounds are optimal in the cell probe model [3] (see also [1]). In Smid's data structure each set is represented by a k-ary balanced tree, where the elements reside in the leaves of the tree.

Kaplan et al. [4] modified Smid's data structure to get a data structure with the same performance for union and find as Smid's, that supports delete in time $O(\log_k n)$ time.

Kaplan et al. also showed how to augment any given data structure that supports *find(x)* in $O(t_f(n))$ worst-case time where n is the size of the set containing x, and insert in time $O(t_i(n))$ worst-case time where n is the size of the set in which we insert the new element, to support *delete(x)* in $O(t_f(n) + t_i(n))$ worst-case time where n is the size of the set containing x, while keeping the same worst-case bounds for union and find. For this general augmentation Kaplan et al. used an incremental rebuilding technique. Each set has two counters, one counting the number of elements in the set, and one counting the number of deleted elements in the set. At each deletion they make a find operation and then increment the number of deleted elements in the set by one. When at least $1/4$ of the elements in a set is deleted they start background rebuilding.

Kaplan et al. also studied the problem in the amortized setting. For the classical union-find problem (wihtout deletions) in the amortized setting Tarjan [7] (and Tarjan and Van Leeuwen [8]) gave a data structure which supports a sequence of M finds and at most N unions on a universe of N elements in $O(N+M\alpha(M+N, N, \log N))$ time where $\alpha(M, N, l) = \min\{k|A_k(\lfloor \frac{M}{N} \rfloor) > l\}$ and $A_i(j)$ is Ackermann's function as defined in [5]. Kaplan et al. refine the analysis of this data structure and show that the cost of each find is proportional to the size of the corresponding set and that it is possible to add a delete operation, such that the amortized cost of deleting an element from a set of size l, is the same as the cost of finding an element, namely $O(\alpha(M, N, \log(l))$.

1.2 Our Results

Kaplan et al. [4] asked if the delete time could be made faster than their find time in the worst case setting. We answer this question affirmatively giving a data structure that supports delete in worst case time $O(\log^* n)$, while keeping the same time bounds for union and find ($O(k)$ and $O(\log_k n)$ respectively).

In this paper $\log n$ denotes the logarithm with base 2. Let $\log^{(i)} n$ denote the logarithm function applied i times in succession. The function $\log^* n$ is defined as $\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$.

We represent the sets as trees with the elements in the leaves, as in the data structure by Smid, and use an incremental rebuilding technique as Kaplan et al., but instead of performing a find operation for every delete operation, we only use $O(\log^* n)$ time on a delete operation. The main idea in our data structure is that we divide the tree into subtrees of certain sizes, and each *delete(x)* operation then perform a kind of path-compression, such that the path to the root gets shorter for the other leaves in the same subtree as x. We do this in a way that ensures, that the leaves in the same subtree helps each other. This is done by only updating parent pointers of the roots in the selected subtrees.

1.3 Overview

The rest of the paper is organized as follows. In Section 2 we give some definitions used in the rest of the paper. In Section 3 we describe the data structure. In Section 4 we give the analysis of the data structure.

2 Definitions

The *height* of a node v, denoted by $h(v)$ is the length of the longest path from v to a leaf in the subtree rooted at v. For a node v let $p(v)$ denote the parent of v.

When we delete a leaf in a tree, we do not actually delete it from the tree, but just mark it as deleted.

The *size* of a tree, denoted by size(T), is the number of leaves in T. If a leaf is marked as deleted it still counts in size(T). Let $|S|$ denote the number of elements in the set S, and let $|T|$ denote the number of the number of leaves in T not marked as deleted.

Definition 1. *Define the function f recursively the following way:*

$$f(0) = 1, \qquad f(i) = 2^{f(i-1)} .$$

We now define the concept *rank* of a node, which plays a crucial part in our data structure.

Definition 2 (Rank of a node). *Let T_v be the subtree rooted at v, and let $m_v = size(T_v)$. The rank r_v of a node v is defined as*

$$r_v = 0 \text{ if } m_v < 16 ,$$

and

$$r_v = 1 \text{ if } 16 \leq m_v < 2^4 \cdot f(3) ,$$

and otherwise

$$r_v = \max\{i : m_v \geq 2^4 \cdot \prod_{j=2}^{i} f(j+1)\} .$$

Or said in another way: If v has rank r_v then the number of leaves in the subtree rooted at v is

$$2^4 \cdot \prod_{j=2}^{r_v} f(j+1) \leq m_v < 2^4 \cdot \prod_{j=2}^{r_v+1} f(j+1) .$$

The rank of a root in a tree of size m is at most $\log^* m + 1$.

At a first glance the definition of rank might look a bit complicated, but the idea behind it is that it has the property that the distance between a node v of rank $i-1$ and its nearest ancestor of rank i or higher is at most $f(i)$. We will prove this in Section 4.

Example 1. In a complete binary subtree all nodes of height less than 4 has rank 0, all nodes of height between 4 and 7 have rank 2, and for $i \geq 2$ all nodes of height between $4+\sum_{j=2}^{i} f(j)$ and $4+\sum_{j=2}^{i+1} f(j)-1$ has rank i.

Define the rank of a (sub)tree to be the rank of its root.

3 The Data Structure

We only describe the data structure for trees with degrees at least 2. That is, the data structure described here use time $O(1)$ for union and $O(\log n)$ for find. The approach can be generalized to k-ary trees, which gives time $O(k)$ for union and $O(\log_k n)$ for find.

We represent each set by trees such that the elements are in the leaves of the tree. Each node contains a pointer to its parent and a linked list of its children. The root of the tree also contains the name of the set, the size of the tree, and the rank of the tree.

We represent each set S by one or two trees. We denote the first tree by S_0 and the second tree if it exists by S_1. We have $|S| = |S_0| + |S_1|$. An element x in S is represented by a leaf in either S_0 or S_1. If x is represented by a leaf in S_0, there might also be a leaf in S_1 associated with x that is marked as deleted. The element x points to the leaf representing it. At the beginning we represent S only by S_0, and S_1 is empty. The tree S_1 will be used when we start rebuilding S_0.

In order to implement the algorithm, we maintain linked lists for S_0 and for S_1, denoted by $L(S_0)$ and $L(S_1)$, respectively. The list $L(S_i)$ contains a node for each leaf in S_i not marked as deleted. The leaf which corresponds to x in S_i points to the node corresponding to x in $L(S_i)$ and vice versa.

From the node identifying S we can get to the nodes identifying S_0 and S_1 and vice versa, and from the node identifying S_i we can get to $L(S_i)$ and vice versa.

3.1 The Operations

Let c be a positive integer constant. We can now define the operations.

find(x): Follow parent pointers from the leaf containing x all the way to the root. Return the name of the set stored at the root.

union(A,B): We perform *Union(A_0,B_0)* and *Union(A_1,B_1)*, where the operation *Union(A,B)* is defined as follows:

Union(A,B): Let a and b be the root of the tree representing A and B respectively. Assume wlog. that size$(A) \geq$ size(B). There are two cases:

1. size$(A) = 1 =$ size(B): Create a new node c, make a and b children of c, and update c's counters.
2. size$(A) > 1$: Make b a child of a, let a be the root of the new set C and update all the counters in a.

Also concatenate $L(A_i)$ with $L(B_i)$ to form $L(C_i)$.

The special case for size$(A) = 1 =$ size(B) ensures that all elements are located at the leaves. Note that the union operation looks at the size of the sets and not the actual number of elements in the set, that is, leaves marked as deleted also counts.

delete(x): Let S be the set containing x. Mark the leaf representing x in S_i as deleted, and delete the node corresponding to x in $L(S_i)$. Follow x's parent pointers up to its first ancestor v of rank $r_v \geq 1$. Now call the following recursive procedure with parameters v and c, (*update(v,c)*).

```
update($v$,$k$)
    • If $v$ is the root call rebuild($S$) (defined below) and exit.
    • If $k = 0$ exit.
    • If $r_{p(v)} > r_v$.
      Call update($p(v)$,$k$) and exit.
    • If $r_{p(v)} = r_v$.
      Update $v$'s parent pointer to point to $p(p(v))$ and call update($v$,$k-1$).
```

The idea behind the delete step is that we mark the node as deleted and send the information along the path to the root. The reason why we only update the parent pointer of certain nodes, is that in this way we ensure, that different delete operations will help each other by updating the same pointers.

rebuild(S): First check whether S_1 is empty, and if this is the case rename S_0 to S_1. Next mark $2c$ leaves in S_1 as deleted (performing $2c$ delete operations without any rebuild steps) and insert the corresponding elements into S_0. We use $L(S_1)$ to find the leaves to be deleted. If S_1 now contains less than $2c$ leaves, then insert the remaining elements into S_0.

4 Analysis

In this section we analyze the data structure. The following lemma follows from the fact that we use a standard union by size operation.

Lemma 1. *The height of a tree T in the data structure is at most $\lceil \log m \rceil$, where $m =$ size(T).*

The following lemma about states a crucial property of the definition of rank.

Lemma 2. *For $i > 1$, the distance between a node v of rank $i-1$ and its nearest ancestor of rank i or higher is at most $f(i)$.*

Proof. The size, size(T_v), of the subtree rooted at the node v is at least $2^4 \cdot \prod_{j=2}^{i-1} f(j+1)$. Wlog. assume that size$(T_v)$ is exactly this amount. By the way we perform union, the number of leaves in the subtree at least doubles every time we follow a parent pointer up. Thus after we have followed

$$2^{2^{\cdot^{\cdot^{2^2}}}} \Big\}^{i} = f(i)$$

parent pointers up, the number of leaves in the subtree rooted at this node is at least $2^4 \cdot \prod_{j=2}^{i} f(j+1)$, and this node thus has rank i or higher. □

As noted earlier the rank of the root of a tree T is at most $\log^* m + 1$, where $m =$ size(T). This gives us the following lemma.

Lemma 3. *Let T be the tree containing the leaf representing the element x. The operation delete(x) takes time $O(\log^* m)$, where $m =$ size(T).*

Proof. If we look at the procedure update without recursive calls and without rebuild it takes constant time.

Going from x to its first ancestor of rank greater than or equal to one takes constant time. We first show that the procedure *update* is called recursively at most $O(\log^* m)$ times during a delete operation.

Update is called recursively when $r_{p(v)} > r_v$. Since the root has rank at most $\log^* m + 1$ this can happen at most $\log^* m + 1$ times. It is also called when $r_{p(v)} = r_v$, but this can only happen c times, since c is decremented by one each time.

Thus, without the call to *rebuild(T) update* takes time $O(\log^* m)$. The procedure *rebuild(T)* is only called once, and it performs $2c$ deletes without any additional calls to *rebuild(T)* which each takes time $O(\log^* m)$, and $2c$ inserts which each takes constant time. □

We now define what we call a *relevant subtree*.

Definition 3. *Let v be a node of rank i. Define the* good *subtree of v, denoted by T_v^*, to be the subtree rooted at v, T_v, with all proper subtrees of T_v of rank i deleted. That is all proper subtrees of T_v^* has rank $i-1$ or less.*

The size *of a good subtree, size(T_v^*), is the number of leaves in T_v^*, not counting v.*

Definition 4. *The* relevant *subtrees of a tree T is the good subtrees of size greater than 0.*

Example 2. A node v is the root of a relevant subtree if it has a child of rank less than itself.

In a complete binary tree T, the relevant subtrees are the subtrees of height exactly $2^4 \cdot \prod_{j=2}^{i} f(j)$ for all $i \leq$ the rank of the root of T, and the distance between the root of a relevant subtree of rank $i-1$ and its nearest ancestor of rank i is exactly $f(i)$.

The relevant subtrees are those subtrees whose root nodes parent pointer might be updated during a delete operation.

The main idea behind the delete operation is that all deletions in the same relevant subtree somehow helps updating the parent pointer of the root of this subtree. We will say that a parent pointer of the root of a relevant subtree of rank i is *fully updated* if it points to its nearest ancestor of rank greater than i. We will later show that when a certain number of leaves of a relevant subtree is deleted, the parent pointer of the root is fully updated.

We need the following lemma.

Lemma 4. *A relevant subtree of rank $i \geq 2$ has at least $2^4 \cdot \prod_{j=2}^{i} f(j+1)$ leaves, and a relevant subtree of rank 1 has at least 16 leaves.*

Proof. By the way we perform union.

The tree is constructed by union operations. Let v be the root of a relevant subtree. Let T_v be the subtree rooted at node v, and let $m_v = \text{size}(T_v)$. Since v is the root of the subtree, then size(T_v) is greater than or equal to the size of the subtrees of the children of v, since we perform union by size. This also means that the rank of v was greater than or equal to the rank of the children of v at the time the union was performed. Thus for all children w of v with rank i, v already had rank i when the *union(T_v,T_w)* was performed. □

An important question in the analysis of the data structure is how many updates do we need to perform before the parent pointers of all relevant subtrees are fully updated, i.e., points to an ancestor of rank greater than the rank of the root itself? Using Lemma 2 we can show the following crucial lemma.

Lemma 5. *Let T be a tree, and r the rank of T. The total number of updates that are needed before the parent pointers of the roots of all the relevant subtrees are fully updated is less than*

$$\frac{1}{3}\textit{size}(T) .$$

When all these parent pointers are fully updated all leaves has distance at most $r+3$ to the root.

Proof. According to Lemma 2 the number of times a parent pointer of rank i needs to be updated is at most $f(i+1)$.

For $i \geq 2$ the number of relevant subtrees of rank i is at most:

$$\frac{\text{size}(T)}{\text{minimum size of a relevant subtree}} \leq \frac{\text{size}(T)}{2^4 \cdot \prod_{j=2}^{i} f(j+1)} ,$$

since the relevant subtrees of rank i are disjoint. The number of relevant subtrees of rank 1 is at most: size$(T)/16$. Thus, the total number of updates needed is at most:

$$\begin{aligned}
&\frac{\text{size}(T)\cdot f(2)}{16} + \sum_{i=2}^{r} f(i+1)\cdot \frac{\text{size}(T)}{2^4\cdot \prod_{j=2}^{i} f(j+1)} \\
&= \frac{\text{size}(T)}{16}\cdot\Big(f(2)+\sum_{i=2}^{r}\frac{f(i+1)}{\prod_{j=2}^{i} f(j+1)}\Big) \\
&= \frac{\text{size}(T)}{16}\cdot\Big(4+\frac{f(3)}{f(3)}+\frac{f(4)}{f(3)\cdot f(4)}+\frac{f(5)}{f(3)\cdot f(4)\cdot f(5)}+\cdots\Big) \\
&= \frac{\text{size}(T)}{16}\cdot\Big(4+1+\frac{1}{f(3)}+\frac{1}{f(3)\cdot f(4)}+\cdots\Big) \\
&< \frac{6\cdot\text{size}(T)}{16} \\
&< \frac{\text{size}(T)}{3}.
\end{aligned}$$

□

To establish the correctness of the algorithm we will show that the fraction of leaves marked as deleted in a tree in the data structure is less than $5/(6c)$.

Lemma 6. *For any set S, if we only perform delete and find operations (no unions) then after less than*

$$\frac{5}{6c}\textit{size}(S_1)$$

delete operations in S_1 the tree S_1 is empty.

Proof. Each deleted operation either give c updates or $2c$ rebuild steps (or maybe both).

If a delete operation calls *rebuild(S)* then we reduce the number of undeleted leaves in S_1 by $2c+1$, since a rebuild moves elements from S_1 to S_0. Thus, in total we need at most size$(S_1)/(2c+1)$ delete operations calling *rebuild(S)*.

By Lemma 5 we can perform less than size$(T)/3$ updates, that is, less than size$(T)/(3c)$ delete operations can result in only updates.

Thus, after less than

$$\frac{\text{size}(T)}{3c}+\frac{\text{size}(T)}{2c+1}<\frac{5}{6c}\text{size}(T)\ .$$

□

Lemma 7. *For any set S, let f be the fraction of leaves marked as deleted in S_1. Then*

$$f<\frac{5}{6c}\ .$$

Proof. By Lemma 6 this is true if we don't perform any unions (and thus also no inserts).

If we perform all the union operations first it is clearly true, but what if we mix them with the deletes?

Since union is performed by size(T) the structure of the final tree (after all unions) looks the same, no matter if we had deletes in between or not.

If we look at the proofs of Lemma 5 and 6, the number of operations needed are given in terms of the total size of the tree, that is, it is also true after the union operation. Since we never “waste” any updates/rebuilds—all delete operations gives at least either c updates or $2c$ rebuilds—it does not matter which order the operations come in. □

Lemma 8. *For every set S, at most $\frac{5}{6c}$ of the elements in S_0 are marked deleted. When at most $\frac{5}{6c}$ of the elements in S_0 are deleted, S_1 is empty.*

Proof. First look at the case where we just have a sequence of delete operations. We will show that if the fraction f of leaves deleted is greater than or equal to $1/(2c)$ and a delete operation calls *rebuild(S)*, then the fraction f' after the delete is at most f. This means that the only way the fraction can grow is if we perform no rebuilds, but only updates.

Assume $f \geq 1/(2c)$ and delete calls *rebuild(S)*. The fraction of deleted elements in S_0 is now

$$\frac{f \cdot \text{size}(S_0) + 1}{\text{size}(S_0) + 2c} = f \cdot \frac{\text{size}(S_0) + 1/f}{\text{size}(S_0) + 2c} \leq f \ .$$

By Lemma 5, then after less than $1/(3c) \cdot \text{size}(S_1)$ updates all parent pointers are updated, and thus all deletes hereafter will call *rebuild(S)*. Note that the insert operations do not make any difference, since insert is equal to union with a tree of size one, and thus all new leaves will be inserted as children of the root, so they do not cause any new updates to be needed.

Thus, the fraction of leaves deleted is less than $1/(2c)+1/(3c) = 5/(6c)$. The only way it can be $5/(6c)$ is if S_1 is empty, and we therefore cannot move any elements from S_1 to S_0. In that case, a rebuild operation will rename S_0 to S_1.

Now look at what happens when we mix the delete operations with unions.

The argument about rebuild keeping the fraction of deleted elements down also holds since, if we perform union on two trees A and B which both have a fraction at least f (or at most f) deleted elements, then the union of the two trees also has a fraction of at least (or most) f deleted leaves.

Again the number of operations needed to update all pointers are given in terms of the total size of the tree, and thus by the same argument as in the proof of Lemma 7 it does not matter which order the operations come in. □

We can now show the main theorem.

Theorem 1. *For any set S we have:*

$$\max\{\textit{size}(S_0), \textit{size}(S_1)\} < (\frac{6c}{6c-5}) \cdot |S| \ .$$

Proof. Let f_1 be the fraction of leaves marked deleted in S_1. By Lemma 7 we have

$$|S_1| = \text{size}(S_1) - f \cdot \text{size}(S_1) \geq (1 - \frac{5}{6c}) \cdot \text{size}(S_1) \ ,$$

and thus

$$\text{size}(S_1) \leq (\frac{6c}{6c-5}) \cdot |S_1| \leq (\frac{6c}{6c-5}) \cdot |S| \ .$$

Let f_0 be the fraction of leaves marked deleted in S_0. By Lemma 8 we have

$$|S_0| = \text{size}(S_0) - f_0 \cdot \text{size}(S_0) \geq (1 - \frac{2}{3c}) \cdot \text{size}(S_0) \ ,$$

and thus

$$\text{size}(S_0) \leq (\frac{3c}{3c-2}) \cdot |S_0| \leq (\frac{3c}{3c-2}) \cdot |S| \ .$$

□

If we set $c \geq 2$ we have that

$$\max\{\text{size}(S_0), \text{size}(S_1)\} < 2 \cdot |S| \ .$$

By Lemma 1 we then have that the height of the trees S_0 and S_1 is at most $\log n$, where n is the number of elements in S. Therefore, the operation *find(x)* takes time $O(\log n)$, where n is the number of elements in the set containing x.

To summarize we have shown the following:

Theorem 2. *Our data structure supports union(A,B) and insert in worst-case $O(1)$ time, find(x) in worst-case $O(\log n)$ time, and delete(x) in worst-case $O(\log^* n)$ time, where n is the size of the set containing x.*

As mentioned earlier we can generalize our approach to k-ary trees, to get worst-case time $O(\log_k n)$ time for find, and worst-case $O(k)$ for union and insert.

References

1. Stephen Alstrup, Amir M. Ben-Amram, and Theis Rauhe. Worst-case and amortised optimality in union-find. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 499–506, New York, May 1999. Association for Computing Machinery.
2. N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. In *Proceedings of the 2nd Annual Symposium on Theoretical Aspects of Computer Science (STACS '85)*, volume 182 of *LNCS*, pages 32–38, Saarbrücken, FRG, January 1985. Springer.
3. Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual Symposium on Theory of Computing (STOC '89)*, pages 345–354, New York, May 1989. ACM Association for Computing Machinery.
4. Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 19–28, New York, January 6–8 2002. ACM Press.
5. D. L. Kozen. *The Design and Analysis of Algorithms*. Springer, Berlin, 1992.
6. M. Smid. A data structure for the union-find problem having good single-operation complexity. *ALCOM: Algorithms Review, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM)*, 1, 1990.
7. R. E. Tarjan. Efficiency of a good but not linear disjoint set union algorithm. *Journal of the ACM*, 22:215–225, 1975.
8. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, April 1984.