Simple Concurrent Connected Components Algorithms

SIXUE CLIFF LIU, Carnegie Mellon University, USA
ROBERT ENDRE TARJAN, Princeton University, USA

We study a class of simple algorithms for concurrently computing the connected components of an n-vertex, m-edge graph. Our algorithms are easy to implement in either the COMBINING CRCW PRAM or the MPC computing model. For two related algorithms in this class, we obtain $\Theta(\lg n)$ step and $\Theta(m \lg n)$ work bounds.[1] For two others, we obtain $O(\lg^2 n)$ step and $O(m \lg^2 n)$ work bounds, which are tight for one of them. All our algorithms are simpler than related algorithms in the literature. We also point out some gaps and errors in the analysis of previous algorithms. Our results show that even a basic problem like connected components still has secrets to reveal.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Shared memory algorithms;**

Additional Key Words and Phrases: Connected components, PRAM algorithms, simplicity

ACM Reference format:
Sixue Cliff Liu and Robert Endre Tarjan. 2022. Simple Concurrent Connected Components Algorithms. *ACM Trans. Parallel Comput.* 9, 2, Article 9 (August 2022), 26 pages.
https://doi.org/10.1145/3543546

1 INTRODUCTION

The problem of finding the connected components of an undirected graph with n vertices and m edges is fundamental in algorithmic graph theory. Any kind of graph search, such as depth-first or breadth-first, solves it in linear time sequentially, which is the best possible way. The problem becomes more interesting in a concurrent model of computation. In the heyday of the theoretical study of **PRAM (parallel random-access machine)** algorithms, many more, and more efficient, algorithms for the problem were discovered, culminating with the $O(\lg n)$ step, $O(m)$ work randomized algorithms of Halperin and Zwick [10, 11], the second of which computes spanning trees of the components. The goal of most of the work in the PRAM model was to obtain the best asymptotic bounds, not the simplest algorithms.

[1]We denote by lg the base-two logarithm.

This paper is a revised and expanded version of [17].
Research at Princeton University partially supported by an innovation research grant from Princeton and a gift from Microsoft. Some work was done at AlgoPARC workshops in 2017 and 2019, partially supported by NSF grants CCF-1930579 and 1745331, respectively.
Authors' addresses: S. C. Liu, Carnegie Mellon University, Pittsburgh, PA, 15213, USA; email: cliffliu@andrew.cmu.edu; R. E. Tarjan, Princeton University, Princeton, NJ, 08540, USA; email: ret@princeton.edu.

2329-4949/2022/08-ART9 $15.00
https://doi.org/10.1145/3543546

With the growth of the internet, the world-wide web, and cloud computing, finding connected components on huge graphs has become commercially important, and practitioners have put versions of the PRAM algorithms into use. Many of the PRAM algorithms are complicated, and even some of the simpler ones have been further simplified when implemented. Experiments suggest that the resulting algorithms perform well in practice, but some of the published claims about their theoretical performance are incorrect or unjustified.

Given this situation, our goal here is to develop and analyze the simplest possible efficient algorithms for the problem and to rigorously analyze their efficiency. In exchange for algorithmic simplicity, we are willing to allow analytic complexity. Our algorithms are easy to implement in either the COMBINING CRCW PRAM model [1] in which write conflicts are resolved in favor of the smallest value, or in the **MPC (massive parallel computing)** model [4].

The COMBINING CRCW PRAM model is stronger than the more standard ARBITRARY CRCW PRAM model, in which write conflicts are resolved arbitrarily (one of the writes succeeds, but the algorithm has no control over which one), but is weaker than the MPC model.

We consider a class of simple deterministic algorithms for the problem. We study in detail four algorithms in the class, all of which are simpler than corresponding existing algorithms. For two of them we prove a step bound of $\Theta(\lg n)$ and a work bound of $\Theta(m \lg n)$. For the other two, we prove a step bound of $O(\lg^2 n)$ and a work bound of $O(m \lg^2 n)$. These bounds are tight for one of the two. We also show that one of our $O(\lg^2 n)$-step algorithms takes $O(d)$ steps where d is the largest diameter of a component, but the others do not.

Our paper is a revised and expanded version of a conference paper [17]. We have deleted an incorrect analysis of one algorithm (algorithm P, Section 4), removed another algorithm for which we had an incorrect analysis, expanded our analysis of a third algorithm (algorithm RA, Section 3), expanded our discussion of related work, and made a number of stylistic changes. The paper contains five sections in addition to this introduction. Section 2 presents our algorithmic framework and a general lower bound in the MPC model. Section 3 presents our algorithms. Section 4 proves upper and lower step bounds. Section 5 discusses related work. Section 6 contains final remarks and open problems.

2 ALGORITHMIC FRAMEWORK

Given an undirected graph with vertex set $[n] = \{1, 2, \ldots, n\}$ and m edges, we wish to compute its connected components via a concurrent algorithm. More precisely, for each component we want to label all its vertices with a unique vertex in the component, so that two vertices are in the same component if and only if they have the same label. To state bounds simply, we assume $n > 2$ and $m > 0$. We denote an edge by the unordered pair of its ends.

As our computing model, we use a COMBINING **CRCW (concurrent read, concurrent write) PRAM (parallel random-access machine)** [1]. Such a machine consists of a large common memory and a number of processes, each with a small private memory. In one step, each process can do one unit of local computation, read one word of common memory, or write into one word of common memory. The processes operate in lockstep. Concurrent reads and writes are allowed, with write conflicts resolved in favor of the smallest value written. We discuss weaker variants of the PRAM model in Section 5. We measure the efficiency of an algorithm primarily by the number of concurrent steps and secondarily by the *work*, defined to be the number of steps times the number of processes.

Our algorithms are also easy to implement on the **MPC (massively parallel computing)** model [4]. This is a model of distributed computation based on the **BSP (bulk synchronous parallel)** model [28]. The MPC model is more powerful than our PRAM model but is a realistic model of cloud computing platforms. An MPC machine consists of a number of processes, each

with a private memory. There is no common global memory; the processes communicate with each other by sending messages. Computation proceeds in globally synchronized steps. In one step, each process receives the messages sent to it in the previous step, does some amount of local computation, and then sends a message or messages to one or more other processes.

We specialize the MPC model to the connected components problem as follows. There is one process per edge and one per vertex. A process can only send a message to another process once it knows about that process. Initially a vertex knows only about itself, and an edge knows only about its two ends. Thus, in the first concurrent step only edges can send messages, and only to their ends. A vertex or edge knows about another vertex or edge once it has received a message containing the vertex or edge. This model ignores contention resulting from many messages being sent to the same process, and it allows a process to send many messages in one step.

The MPC model is quite powerful, but even in this model there is a non-constant lower bound on the number of steps needed to compute connected components.

Theorem 2.1. *Computing connected components in the MPC model takes* $\Omega(\lg d)$ *steps, where* d *is the largest diameter of a component.*

Proof. Let u be the vertex that eventually becomes the label of all vertices in a component of diameter d. Some vertex v is at distance at least $d/2$ from u. An induction on the number of steps shows that after k steps a vertex has only received messages containing vertices within distance 2^{k-1}. Since v must receive a message containing u, the computation takes at least $\lg d$ steps. □

It is easy to solve the problem in $O(\lg d)$ steps if messages can be arbitrarily large: send each edge end to the other end, and then repeatedly send from each vertex all the vertices it knows to all its incident vertices [21]. If there is a large component, however, this algorithm is not practical, for at least two reasons: it requires huge memory at each vertex, and the number of messages sent in the last step can be quadratic in n. Hence, we restrict the local memory of a vertex or edge to hold only a small constant number of vertices and edges. We also restrict messages to hold only a small constant number of vertices and edges, along with an indication of the message type, such as a label request or a label update. Our goal is a simple algorithm with a step bound of $O(\lg n)$. (We discuss the harder goal of achieving a step bound of $O(\lg d)$ in Section 5.)

We consider algorithms that maintain a label for each vertex u, initially u itself. The algorithm updates labels step-by-step until none changes, after which all vertices in a component have the same label, which is one of the vertices in the component. At any given time the current labels define a **digraph (directed graph)** of out-degree one whose arcs lead from vertices to their labels. We call this the *label digraph*. If these arcs form no cycles other than loops (arcs of the form (u, u)), then this digraph is a forest of trees rooted at the self-labeled vertices: the parent of u is its label unless this label is u, in which case u is a root. We call this the *label forest*. Each tree in the forest is a *label tree*.

All our algorithms maintain the label digraph as a forest; that is, they maintain acyclicity except for self-labels. (We know of only two previous algorithms that do not maintain the label digraph as a forest: see Section 5.) Henceforth, we call the label of a vertex u its *parent* and denote it by $u.p$, and we call a label tree just a *tree*. A non-root vertex is a *child* of its parent. A vertex is a *leaf* if it is a child but it has no children of its own. A tree is *flat* if the root is the parent of every child in the tree, and is a *singleton* if the root is the only vertex. (Some authors call a flat tree a *star*.) The *depth* of a tree vertex x is the number of arcs on the path from x to the root of the tree: a root has depth zero, a child of a root has depth one. The *depth of a tree* is the maximum of the depths of its vertices.

When changing labels, a simple way to guarantee acyclicity is to replace a parent only by a smaller vertex. We call this *minimum labeling*. (An equivalent alternative, *maximum labeling*, is to replace a parent only by a larger vertex). A minimum labeling algorithm stops with each vertex labeled by the smallest vertex in its component. All our algorithms do minimum labeling. They also maintain the invariant that all vertices in a tree are in the same component, as do all algorithms known to us. That is, they never create a tree containing vertices from two or more components. Our specialization of the MPC model to the connected components problem maintains this invariant. At the end of the computation there is one flat tree per component, whose root is the minimum vertex in the component.

3 ALGORITHMS

We consider algorithms that consist of initialization followed by a main loop that updates parents and repeats until no parent changes. Initialization consists of setting the parent of each vertex equal to itself. The following pseudocode does initialization:

initialize:
  for each vertex v **do** $v.p = v$

Since initialization is the same for all our algorithms, we omit it in the descriptions below and focus on the main loop. Each iteration of the loop does a *connect* step, which updates parents using current edges, one or more *shortcut* steps, each of which updates parents using old parents, and possibly an *alter* step, which alters the edges of the graph. When discussing and analyzing our algorithms, we use the following terminology. By the *graph* we mean the input graph, or the graph formed from the input graph by the *alter* steps done so far, if the algorithm does *alter* steps. An *edge* is an edge of the graph; a *component* is a connected component of the graph. All *alter* steps preserve components. By the *forest* we mean the forest whose child-parent arcs are the pairs $(v, v.p)$ with $v \neq v.p$; a *tree* is a tree in this forest. The terms *parent*, *child*, *ancestor*, *descendant*, *root*, and *leaf* refer to the forest.

Our algorithms maintain the following *connectivity invariant*: v and $v.p$ are in the same component, as are v and w if $\{v, w\}$ is an edge. If $\{v, w\}$ is an edge, replacing the parent of v by any ancestor of w preserves the invariant, as does replacing the parent of w by any ancestor of v. This gives us many ways of doing a connect step. We focus on two. The first is *direct-connect*, which for each edge $\{v, w\}$, uses the minimum of v and w as a candidate for the new parent of the other. The other is *parent-connect*, which uses the minimum of the old parents of v and w as a candidate for the new parent of the old parent of the other. We express these methods in pseudocode below. We have written all our pseudocode so that it is correct and produces unambiguous results even if the loops run sequentially rather than concurrently and the vertices and edges are processed in arbitrary order. We say more about this issue below.

direct-connect:
  for each edge $\{v, w\}$ **do**
    if $v > w$ **then**
      $v.p = \min\{v.p, w\}$
    else $w.p = \min\{w.p, v\}$

The pseudocode for *parent-connect* begins by computing $v.o$, the old parent of v, for each vertex v. It then uses these old parents to compute the new parent $v.p$ of each vertex v. The new parent of a vertex x is the minimum $w.o < x.o$ such that there is an edge $\{v, w\}$ with $v.o = x$, if there is such an edge; if not, the parent of x does not change.

parent-connect:
  for each vertex v **do** $v.o = v.p$
  for each edge $\{v, w\}$ **do**
    if $v.o > w.o$ **then**
      $v.o.p = \min\{v.o.p, w.o\}$
    else $w.o.p = \min\{w.o.p, v.o\}$

If all reads and comparisons occur before all writes, and all writes occur concurrently, the following simpler pseudocode has the same semantics as that for *parent-connect*:

for each edge $\{v, w\}$ **do**
  if $v.p > w.p$ **then**
    $v.p.p = \min\{v.p.p, w.p\}$
  else $w.p.p = \min\{w.p.p, v.p\}$

On the other hand, if this simpler loop is executed sequentially, then in general the results depend on the order in which the edges are processed. Suppose for example there are two edges $\{x, y\}$ and $\{v, w\}$ such that $x.p = v$ and $v.p = z$. In *parent-connect*, $w.p$ is a candidate to be the new parent of z. That is, after the connect, the new parent of z will be no greater than the old parent of w. But in the simpler loop, if $\{x, y\}$ is processed before $\{v, w\}$, the processing of $\{x, y\}$ might change the parent of v to a vertex other than z, thereby making $w.p$ no longer a candidate for the new parent of z. Even though we are primarily interested in global concurrency, we want our algorithms and bounds to be correct in the more realistic setting in which the edges are processed one group at a time, with the group size determined by the number of available processes.

On a COMBINING CRCW PRAM, we can use the simple loop for *parent-connect*, since there is global concurrency. Each process for an edge $\{v, w\}$ reads $v.p$ and $w.p$. If $v.p > w.p$, it reads $v.p.p$, tests if $w.p < v.p.p$; and, if so, writes $w.p$ to $v.p.p$; if $v.p \leq w.p$, it reads $w.p.p$, tests if $v.p < w.p.p$; and, if so, writes $v.p$ to $w.p.p$. All the reads occur before all the writes, and all the writes occur concurrently, with a write of smallest value succeeding if there is a conflict.

In the MPC model, each vertex stores its parent. To execute *parent-connect*, each process for an edge $\{v, w\}$ requests $v.p$ and $w.p$. If $v.p > w.p$, it sends $w.p$ to $v.p$; otherwise, it sends $v.p$ to $w.p$. Each vertex then updates its parent to be the minimum of its old value and the smallest of the received values. All our other loops can be similarly implemented on a COMBINING CRCW PRAM or in the MPC model.

A connect step of either kind can move a subtree from one tree to another. We can prevent this by restricting connection so that it only updates parents of roots. The following pseudocode implements such restrictions of *direct-connect* and *parent-connect*, which we call *direct-root-connect* and *parent-root-connect*, respectively:

direct-root-connect:
  for each vertex v **do** $v.o = v.p$
  for each edge $\{v, w\}$ **do**
    if $v > w$ and $v = v.o$ **then**
      $v.p = \min\{v.p, w\}$
    else if $w = w.o$ **then**
      $w.p = \min\{w.p, v\}$

parent-root-connect:
  for each vertex v **do** $v.o = v.p$
  for each edge $\{v, w\}$ **do**
    if $v.o > w.o$ and $v.o = v.o.o$ **then**
      $v.o.p = \min\{v.o.p, w.o\}$
    else if $w.o = w.o.o$ **then**
      $w.o.p = \min\{w.o.p, v.o\}$

In *direct-root-connect* (as in *parent-connect*) we need to save the old parents to get a correct sequential implementation, so that the root test is correct even if the parent has been changed by processing another edge during the same iteration of the loop over the edges. If we truly have global concurrency, simpler pseudocode suffices, as for *parent-connect*.

Shortcutting is the key to obtaining a logarithmic step bound. Shortcutting replaces the parent of each vertex by its grandparent. The following pseudocode implements shortcutting:

shortcut:
  for each vertex v **do**
    $v.o = v.p$
  for each vertex v **do**
    $v.p = v.o.o$

In the case of *shortcut*, the simpler loop "for each vertex v do $v.p = v.p.p$" produces correct results and preserves our time bounds, even though sequential execution of the code produces different results depending on the order in which the vertices are processed. All we need is that the new parent of a vertex is no greater than its old grandparent. Thus, the simpler loop might well be a better choice in practice.

Edge alteration deletes each edge $\{v, w\}$ and replaces it by $\{v.p, w.p\}$ if $v.p \neq w.p$. The following pseudocode implements alteration:

alter:
  for each edge $\{v, w\}$ **do**
    if $v.p = w.p$ **then**
      delete $\{v, w\}$
    else replace $\{v, w\}$ by $\{v.p, w.p\}$

We shall study in detail four algorithms, whose main loops are given below:

Algorithm S: **repeat** {*parent-connect*; **repeat** *shortcut* **until** no $v.p$ changes} **until** no $v.p$ changes

Algorithm R: **repeat** {*parent-root-connect*; *shortcut*} **until** no $v.p$ changes

Algorithm RA: **repeat** {*direct-root-connect*; *shortcut*; *alter*} **until** no $v.p$ changes

Algorithm A: **repeat** {*direct-connect*; *shortcut*; *alter*} **until** no $v.p$ changes

In algorithm S, the inner loop "**repeat** *shortcut* **until** no $v.p$ changes" terminates after a shortcut that changes no parents. Similarly, in each of the algorithms the outer **repeat** loop terminates after an iteration that changes no parents.

Many algorithms fall within our framework. We focus on these four because they are simple and natural and we can prove good bounds for them. Algorithm S simplifies the algorithm of

Hirschberg, Chandra, and Sarwate [12]. Theirs was the first algorithm to run in a polylogarithmic number of steps, specifically $O(\lg^2 n)$. Algorithm R simplifies the algorithm of Shiloach and Vishkin, which runs in $O(\lg n)$ steps. We discuss in detail the relationship of our algorithms to theirs, as well as other related work, in Section 5.

Some observations guided our choice of algorithms to study. There is a tension between connect steps and shortcut steps: the former combine trees but generally produce deeper trees; the latter make trees shallower. Algorithm S completely flattens all the existing trees between each connect step. This guarantees monotonicity: the *parent-connect* steps in S change the parents only of roots. As we shall see, completely flattening the trees also guarantees that each tree is connected to another tree in at most two iterations of the outer loop. Algorithm S has an $O(\lg^2 n)$ step bound, one log factor coming from the at most $\lg n$ iterations of the inner loop needed to flatten all trees, the other coming from the at most $2lgn$ iterations of the outer loop needed to reduce the number of trees in each component to 1. Unfortunately, the $O(\lg^2 n)$ bound is tight to within a constant factor.

To obtain a step bound of $\lg n$, we must reduce the number of shortcuts per round to $O(1)$. Algorithm R does this. It uses *root-connect*, making it monotonic. Algorithm RA is a related algorithm that does edge alteration, allowing it to use *direct-root-connect* instead of the more-complicated *root-connect*. Both R and RA have an $O(\lg n)$ step bound.

It is natural to wonder whether monotonicity is needed to get a polylogarithmic step bound. Algorithm A answers this question negatively. It is algorithm RA with *direct-connect* replacing *direct-root-connect*. We shall prove an $O(\lg^2 n)$ step bound for algorithm A. We do not know if this bound is tight.

Each of our four algorithms is distinct: for each pair of algorithms, there is a graph on which the two algorithms make different parent changes, as one can verify case-by-case. Use of *direct-connect* or *direct-root-connect* requires edge alteration to obtain a correct algorithm. (A counterexample is the graph whose vertices are $1, 2, 3$ and whose edges are $\{1, 3\}, \{2, 3\}$: the parent of vertex 2 remains 2 but should become 1.) Although different, algorithms R and RA behave similarly, and we use the same techniques to obtain bounds for both of them. Algorithm S is equivalent to the algorithm formed by replacing *parent-connect* by *parent-root-connect*, and to the algorithm formed by replacing *parent-connect* by *direct-connect* and adding *alter* to the end of the main loop: all three algorithms make the same parent changes.

We conclude this section with a proof that our algorithms are correct. We begin by establishing some properties of edge alteration. We call an iteration of the main loop (the outer loop in algorithm S) a *round*. We call a vertex *bare* if it is not an edge end and *clad* if it is. Only alter steps can change vertices from clad to bare or vice-versa. A clad vertex becomes bare if it loses all its incident edges during an alter step. A bare leaf cannot become clad, nor can a bare vertex all of whose descendants are bare, because no connect step can give it a new (clad) descendant.

To prove correctness, we need the following key result.

Lemma 3.1. *After k rounds of algorithm A or RA, each vertex that is clad or a root has a path in the graph to the smallest vertex in its component. If the algorithm is A, there is such a path with at most* $\max\{0, d - k\}$ *edges.*

Proof. The proof is by induction on k. The lemma is true for $k = 0$. Let $k > 0$, let x be a clad vertex or a root at the end of round k, and let w be the smallest vertex in the same component as x. If $x = w$, the lemma holds for x and k. Suppose $x > w$. If x is a root just before the alteration in round k, let $u = x$; otherwise, let u be a vertex such that $u.p = x$ and u is clad just before the alteration in round k. Such a u must exist: x is not a root at the end of round k and hence is clad at the end of round k; x must have become an edge end during the alteration at the end of round k by the alteration of an edge (u, v) with $u.p = x$. Since $u > x > w$, $u \neq w$. By the induction

hypothesis, there is a path in the graph from u to w at the beginning of round k, and if the algorithm is A, this path contains at most $\max\{0, d-k+1\}$ edges. Let $\{v, w\}$ be the last edge on this path. The alteration in round k converts this path to a path from x to w. Since $\{v, w\}$ exists at the beginning of round k, $v.p = w$ after the connect in round k. Thus, the alter in round k deletes $\{v, w\}$, so if the algorithm is A, the path from x to w contains at most $\max\{0, d-k\}$ edges. □

Lemma 3.2. *In algorithms A and RA,* (i) *an alter makes all leaves bare;* (ii) *once a vertex is a leaf, it stays a leaf;* (iii) *once a leaf is bare, it stays bare; and* (iv) *every non-root grandparent is clad.*

Proof. Part (i) is immediate from the definition of alter. A vertex becomes a leaf either in a connect or in a shortcut. A shortcut does not make a leaf into a non-leaf. Any leaf existing at the beginning of a connect is bare by (i) and hence, cannot be made a non-leaf by the connect. Neither a connect nor a shortcut can make a bare vertex clad. Thus, (ii) and (iii) hold.

We prove (iv) by induction on the number of steps. No vertex is a grandparent initially, so (iv) holds initially. Let x be a vertex that is not a non-root grandparent before a connect; that is, x is either a root or has no grandchildren. If x acquires a parent or child during the connect, then it must be clad, since *direct-connect* only changes parents and children of clad vertices, so the lemma holds for x after the connect. Since all children of x (if any) are leaves, they cannot become non-leaves during the connect by (i), so x cannot become a grandparent as a result of one of its children acquiring a child. Thus, the connect preserves (iv). A vertex that is a non-root grandparent after a shortcut is also a non-root grandparent before the shortcut, so a shortcut preserves (iv). Suppose x is a non-root grandparent before an alter. Then x is a great-great grandparent before the shortcut preceding the alter, so it has a non-root grandchild y that is clad. The shortcut makes y a child of x. By Lemma 3.1 there is a path in the graph from y to the smallest vertex in its component. The alter transforms this path into a path from x to the smallest vertex. Since x is not a root, it is not the smallest vertex, so the path from x contains at least one edge, making x clad. □

Lemma 3.3. *In all our algorithms, a leaf stays a leaf.*

Proof. For A or RA, this is part (iii) of Lemma 3.2. For the other algorithms, a connect or shortcut cannot make a leaf into a parent. □

Theorem 3.4. *All our algorithms are correct.*

Proof. For each algorithm, a proof by induction on the number of parent changes and alterations (if any) shows that v and $v.p$ are in the same component of the input graph for each vertex v, as are v and w for each edge $\{v, w\}$. Thus, all vertices in any tree are in the same component.

Parents only decrease, so the parent function always defines a forest. There are at most $n(n-1)/2$ parent changes. In the following paragraph, we prove that there is at least one parent change in any round except the last one, so each algorithm terminates in at most $n(n-1)/2 + 1$ rounds.

Consider the state just before a round. If there is at least one non-flat tree, then either the connect step or the first shortcut step will change a parent. Suppose all trees are flat but the vertices in some component are in two or more trees. Let T be the tree containing the smallest vertex in the component and T' another tree in the component. It is immediate for algorithms S and R and follows from Lemma 3.1 for A and RA that there is a path in the graph from the root of T' to the root of T. Thus, there is an edge $\{v, w\}$ with w but not v in T. Since all trees are flat, $v.p$ and $w.p$ are roots. In algorithms A and RA, v and w are roots by part (i) of Lemma 3.2. In algorithms S and R, the connect step will make $w.p$ the parent of $v.p$; in algorithms A and RA, the connect step will make w the parent of v. We conclude that the algorithm can only stop when all trees are flat and there is one tree per component. □

4 EFFICIENCY

On a graph that consists of a path of n vertices, all our algorithms take $\Omega(\lg n)$ steps, since this graph has one component of diameter $n-1$, and the general lower bound of Theorem 2.1 applies. We prove worst-case step bounds of $O(\min\{d, \lg n\} \lg n)$ for S, $O(\min\{d, \lg^2 n\})$ for A, and $O(\lg n)$ for R and RA. We also show that algorithm S can take $\Omega(\lg^2 n)$ steps, and algorithms R and RA can take $\Omega(\lg n)$ steps even on graphs with constant diameter d.

4.1 Analysis of S

Theorem 4.1. *Algorithm S takes $O(\min\{d, \lg n\} \lg n)$ steps.*

Proof. Since the maximum depth of any tree is $n-1$, and a shortcut reduces the depth of a depth-k tree to $\lceil k/2 \rceil$, the inner loop stops in $O(\lg n)$ iterations. Let u be the smallest vertex in some component. We prove by induction on k that after k rounds every vertex in the component at distance k or less from vertex u has parent u, which implies that the algorithm stops in at most $d+1$ rounds. This is true initially. Let v be a vertex at distance k from u. If $v.p \neq u$ just before round k, there is an edge $\{v, w\}$ such that $w.p = u$ by the induction hypothesis. After the connect step in round k, $v.p = u$.

To obtain an $O(\lg n)$ bound on the number of rounds, we prove that if there are two or more trees in a component, two rounds reduce the number of such trees by at least a factor of two. Call a root *minimal* if edges incident to its tree connect it only with trees having higher roots. A connect step makes each non-minimal root into a non-root. Let x be a minimal root, and let $\{v, w\}$ be an edge with v in the tree rooted at x and w in a tree rooted at $y > x$. If the connect step in the round does not make y a child of x, then $y.p < x$ after the round, causing x to become a non-root in the next round.

Suppose there are $k > 1$ roots in a component at the beginning of a round. Among the minimal roots, suppose there are j that get a new child as a result of the connect step in the round. At least $\max\{k-j, j\} \geq k/2$ roots are non-roots after two rounds. □

We show by example that S can take $\Omega(\lg^2 n)$ steps.

If there is a tree of depth 2^k just before the inner loop in a round, this loop will take at least k steps. If every round produces a *new* tree of depth 2^k, and the algorithm takes k rounds, the total number of steps will be $\Omega(k^2)$. Our bad example is based on this observation. It consists of k components such that running algorithm S for i rounds on the i-th component produces a tree that contains a path of length 2^k.

At the beginning of a round of the algorithm, we define the *implied graph* to be the graph whose vertices are the roots and whose edges are the pairs $\{v.p, w.p\}$ such that $\{v, w\}$ is a graph edge and $v.p \neq w.p$. Restricted to the roots, the subsequent behavior of algorithm S on the original graph is the same as its behavior on the implied graph. We describe a way to produce a given implied graph in a given number of rounds.

To produce a given implied graph G with vertex set $[n]$ in one round, we start with the *generator* $g(G)$ of G, defined to be the graph with vertex set $[2n]$, edges $\{v, v+n\}$ and $\{v+n, v\}$ for each $v \in [n]$, and an edge $\{v+n, w+n\}$ for each edge $\{v, w\}$ in G. A round of algorithm S on $g(G)$ does the following: the connect step makes $v+n$ a child of v for $v \in [n]$, and the shortcuts do nothing. The resulting implied graph has vertex set $[n]$ and an edge $\{v, w\}$ for each edge $\{v, w\}$ in G; that is, it is G.

To produce a given implied graph G in i rounds, we start with $g^i(G)$. An induction on the number of rounds shows that after i rounds of algorithm S, G is the implied graph.

Let k be a positive integer, and let P be the path of vertices $1, 2, \ldots, 2^k + 1$ and edges $\{i, i+1\}$ for $i \in [2^k]$. Our bad example is the disjoint union of $P, g(P), g^2(P), \ldots, g^{k-1}(P)$, with the vertices renumbered so that each component has distinct vertices and the order within each component is preserved. Algorithm S takes $\Omega(k^2)$ steps on this graph. The number of vertices is $n = (2^k + 1)(2^k - 1) = 2^{2k} - 1$. The number of edges is $2^{2k} - k - 1$, since the graph is a set of k trees. Thus, the number of steps is $\Omega(\lg^2 n)$.

4.2 Analysis of A

In analyzing A, R, and RA, we assume that the graph is connected: this is without loss of generality since A, R, and RA operate independently and concurrently on each component, and each round does $O(1)$ steps.

Theorem 4.2. *Algorithm A takes $O(d)$ steps.*

Proof. By Lemma 3.1, after d rounds there are no edges and only one tree. By Lemma 3.2, this tree has depth at most two. Thus, the algorithm stops after at most $d + 2$ rounds. □

A simple example shows that Theorem 4.2 is false for S, R, and RA. Consider the graph whose edges are $\{i, i+1\}$ for $i \in [n-1]$ and $\{i, n\}$ for $i \in [n-1]$. After the first connection step, there is one tree: 1 is the parent of 2 and n, and i is the parent of $i+1$ for $i \in [2, n-2]$. Subsequent connection steps do nothing; the tree only becomes flat after $\Omega(\lg n)$ shortcuts.

To obtain an $O(\lg^2 n)$ step bound for algorithm A, we show that $O(\lg n)$ rounds reduce the number of non-leaf vertices by at least a factor of 2, from which an overall $O(\lg^2 n)$ step bound follows.

It is convenient to shift our attention from rounds to passes. A *pass* is the interval from the beginning of one shortcut to the beginning of the next. Pass 1 begins with the shortcut in round 1 and ends with the connect in round 2. We need one additional definition. A vertex is *deep* if it is a non-root with at least one child and all its children are leaves.

Lemma 4.3. *A vertex that is deep at the beginning of a pass is a leaf at the end of the pass.*

Proof. Let x be a vertex that is deep at the beginning of a pass. The shortcut in the pass makes x a leaf. Once a vertex is a leaf, it stays a leaf by Lemma 3.2. □

Lemma 4.4. *Suppose there are at least two roots at the beginning of a pass, and that x is a root all of whose children are bare leaves. Then x is not a root after the pass.*

Proof. By Lemma 3.1, at the beginning of the pass there is an edge $\{v, w\}$ with v but not w in the tree with root x. Since all children of x are bare leaves, $v = x$. The edge $\{x, w\}$ existed at the beginning of the connect just before the pass. Since this connect did not make x a non-root, $w > x$, and since w is not a child of x after the connect, $w.p < x$ after it. The alter in the pass replaces $\{x, w\}$ by $\{x, w.p\}$. The connect in the pass then makes x a non-root. □

We need one more idea, which we borrow from the analysis of *path halving*, a method used in disjoint set union algorithms that shortcuts a single path [27]. For any vertex v, we define the *level* of v to be $v.l = \lfloor \lg(v - v.p) \rfloor$ unless $v.p = v$, in which case $v.l = 0$. The level of a vertex is at least 0, less than $\lg n$, and non-decreasing. The following lemma quantifies the effect of a shortcut on a sufficiently long path in a tree.

Lemma 4.5. *Assume $n \geq 4$. Consider a tree path P of $k \geq 4 \lg n$ vertices. A shortcut increases the sum of the levels of the vertices on P by at least $k/4$.*

Proof. Let u, v, and w be three consecutive vertices on P, with v the parent of u and w the parent of v. Let i and j be the levels of u and v, respectively. A shortcut increases the level of u from i to $\lfloor \lg(u - w) \rfloor = \lfloor \lg(u - v + v - w) \rfloor \geq \lfloor \lg(2^i + 2^j) \rfloor$. If $i < j$, this increases the level of u by at least $j - i$; if $i = j$, it increases the level of u by one.

Let $x_1, x_2, \ldots, x_{k-1}$ be the vertices on P from largest to smallest (deepest to shallowest), excluding the last one. For each $i \in [k-2]$, let $\Delta_i = x_{i+1}.l - x_i.l$. The sum of the Δ_i's is $\Sigma = x_{k-1}.l - x_1.l \geq -\lg n$ since the sum telescopes. Let k_+, k_0, and k_-, respectively, be the number of positive, zero, and negative Δ_i's, and let Σ_+ and Σ_- be the sum of the positive Δ_i's and the sum of the negative Δ_i's, respectively. By the previous paragraph, the sum of the levels of the vertices on P increases by at least $\Sigma_+ + k_0$.

From $\Sigma = \Sigma_+ + \Sigma_-$ we obtain $\Sigma_+ \leq -\Sigma_- - \lg n$. Since the Δ_i's are integers, $\Sigma_+ \geq k_+$ and $-\Sigma_- \geq k_-$. Thus $2(\Sigma_+ + k_0) \geq 2\Sigma_+ + k_0 \geq k_+ + k_0 + k_- - \lg n = k - 2 - \lg n \geq k/2$, since $n \geq 4$ implies $k/2 \geq 2\lg n \geq \lg n + 2$. Dividing by two gives $\Sigma_+ + k_0 \geq k/4$. □

We combine Lemmas 4.3, 4.4, and 4.5 to obtain the desired result.

Lemma 4.6. *Assume $n \geq 4$. Suppose that at the beginning of pass i there are at least two roots and more than $k/2$ but at most k non-leaves. After $O(\lg n)$ passes there are at most $k/2$ non-leaves or at most one root.*

Proof. Assume the hypotheses of the lemma are true. Call a vertex *fresh* if it is a non-leaf at the beginning of pass i and *stale* otherwise. After pass i, all the stale leaves are bare by Lemma 3.2. Suppose the hypotheses of the lemma hold at the beginning of some pass after pass i. At least one of the following four cases occurs:

(1) There are at least $k/8$ clad leaves. One pass makes all clad leaves bare by Lemma 3.2. Since each clad leaf is fresh and can only become bare once, there are at most $k/(k/8) = 8$ passes in which this case can occur.
(2) There are at least $k/8$ roots of flat trees, all of whose children are bare. One pass makes all but one such roots non-roots by Lemma 4.4. There are at most $k/(k/8) = 8$ passes in which this case can occur.
(3) There are at least $k/(32\lg n)$ deep vertices. This pass makes all these vertices into leaves by Lemma 4.3. There are at most $k/(k/(32\lg n)) = 32\lg n$ passes in which this case can occur.
(4) None of the first three cases occurs. Since Cases 1 and 2 do not occur, there are at most $k/4$ roots of flat trees. Thus there are at least $k/4$ non-leaves in non-flat trees. Since Case 3 does not occur, there are at most $k/(32\lg n)$ deep vertices. From each of these deep vertices there is a tree path to a root. Every non-leaf in a non-flat tree is on one or more such paths. Find a deep vertex whose tree path is longest, and delete this path. This may break the tree containing the path into several trees, but this does not matter: it does not increase the number of deep vertices, although it may convert some deep vertices into roots, which only improves the bound. Repeat this process until at least $k/8$ non-leaves are on deleted paths. Each path contains at least $(k/8)/(k/(32\lg n)) = 4\lg n$ vertices. By Lemma 4.5, the shortcut increases the sum of the levels of the vertices on these paths by at least $k/32$. There are at most $(k\lg n)/(k/32) = 32\lg n$ passes in which this case can occur.

We conclude that after at most $16 + 64\lg n$ passes, either there are at most $k/2$ non-leaves or at most one root. □

Theorem 4.7. *Algorithm A takes $O(\lg^2 n)$ steps.*

Proof. The theorem is immediate from Lemma 4.6, since once there is a single root the algorithm stops after $O(\lg n)$ steps. □

We do not know whether the bound in Theorem 4.7 is tight. We conjecture that it is not, and that algorithm A takes $O(\lg n)$ steps. We *are* able to prove an $O(\lg n)$ bound for the monotone algorithms R and RA, which we do in the next section.

An algorithm similar to algorithm A is algorithm P, which replaces *direct-connect* in A by *parent-connect* and deletes *alter*. The following pseudocode implements the main loop of this algorithm:

Algorithm P: **repeat** {*parent-connect*; *shortcut*} **until** no $v.p$ changes

We conjecture that algorithm P, too, has an $O(\lg n)$ step bound, but we are unable to prove even an $O(\lg^2 n)$ bound, the problem being that this algorithm can leave flat trees unchanged for a non-constant number of rounds.

4.3 Analysis of R and RA

Algorithms R and RA can also leave flat trees unchanged for a non-constant number of rounds, so the analysis of Section 4.2 also fails for these algorithms. But we can obtain an even better bound than that of Theorem 4.7 by using a different analytical technique, that of Awerbuch and Shiloach [3], extended to cover a constant number of rounds rather than just one. This analysis requires the algorithm to be monotonic.

We call a tree *passive* in a round if it exists both at the beginning and at the end of the round; that is, the round does not change it. A passive tree is flat, but a flat tree need not be passive. We call a tree *active* in a round if it exists at the end of the round but not at the beginning. An active tree contains at least two vertices and has depth at least one.

We say a connect *links* trees T and T' if it makes the root of one of them a child of a vertex in the other. If the connect makes the root of T a child of a vertex in T', we say the connect *links* T *to* T'.

Lemma 4.8. *If trees T and T' are passive in round k, then there is no edge with one end in T and the other end in T', and the connect in round $k+1$ does not link T and T'.*

Proof. If T and T' were linked in round $k+1$, there would be an edge connecting them that caused the link. In algorithm RA, Lemma 3.2 implies that any such edge connects the roots of T and T' at the beginning of round k. Thus, in either algorithm T and T' would have been linked in round k, contradicting their passivity in round k. □

If T exists at the end of round k, its *constituent trees* at the end of round $j \leq k$ are the trees existing at the end of round j whose vertices are in T. Since algorithms R and RA are monotone, these trees partition the vertices of T.

Lemma 4.9. *Let T be an active tree in round k. Then for $j \leq k$ at least one of the constituent trees of T in round j is active.*

Proof. The proof is by induction on j for j decreasing. The lemma holds for $j = k$ by assumption. Suppose it holds for $j > 0$. If the constituent trees of T in round $j-1$ were all passive, the connect step in round j would change none of them by Lemma 4.8. Neither would the shortcut in round j, contradicting the existence of an active constituent tree in round j. □

Since algorithm RA is slightly simpler to analyze than R, we first analyze RA, and then discuss the changes needed to make the analysis apply to R. We measure progress using the potential function of Awerbuch and Shiloach, modified so that it is non-increasing and passive trees have zero potential. In RA, we define the *individual potential* of a tree T at the end of round k to be zero

if T is passive in round k, or two plus the maximum of zero and the maximum depth of an arc end in T if T is active in round k. If T exists at the end of round k and $j \leq k$, we define the *potential* $\Phi_j(T)$ of T at the end of round j to be the sum of the individual potentials of its constituent trees at the end of round j. We define the *total potential* at the end of round k to be the sum of the potentials of the trees existing at the end of round k.

We shall prove that the total potential decreases by a constant factor in a constant number of rounds. This will give us an $O(\lg n)$ bound on the number of rounds. It suffices to consider each active tree individually.

Lemma 4.10. *Let T be active in round $k > 1$ of RA. Then $\Phi_{k-1}(T) \geq \Phi_k(T)$. If $\Phi_{k-1}(T) \geq 4$, then $\Phi_{k-1}(T) \geq (5/4)\Phi_k(T)$.*

Proof. Let $t \geq 1$ be the number of active constituent trees of T in round $k-1$, and let $\ell = \Phi_{k-1}(T) - 2t$. Then $\ell \geq 0$. Consider the tree S formed from the constituent trees of T in round $k-1$ by the connect step in round k. The shortcut in round k transforms S into T. By Lemma 4.8, along any path in S there cannot be consecutive vertices from two different passive trees. By Lemma 3.2, at the beginning of round k no edge end is a leaf, so the deepest edge end in S has depth at most $\ell + 2t$: its path to the root contains at most $\ell + t$ vertices in active constituent trees and at most $t+1$ vertices (all roots) in passive constituent trees. The shortcut of S in round k reduces the maximum depth of an arc end to at most $\lceil \ell/2 \rceil + t$. The alter in round k reduces this maximum depth by at least one, to at most $\lceil \ell/2 \rceil + t - 1$. Thus $\Phi_k(T) \leq \lceil \ell/2 \rceil + t - 1$. Since $\Phi_{k-1}(T) = \ell + 2t$ and $t \geq 1$, $\Phi_{k-1}(T) \geq \Phi_k(T)$, giving the first part of the lemma.

We prove the second part of the lemma by induction on $\Phi_{k-1}(T) = \ell + 2t$. If $\Phi_{k-1}(T) = 4$, then $t = 2$ and $\ell = 0$, or $t = 1$ and $\ell = 2$, so $\Phi_k(T) \leq \lceil \ell/2 \rceil + t + 1 = 3$. If $\Phi_{k-1}(T) = 5$, then $t = 2$ and $\ell = 1$, or $t = 1$ and $\ell = 3$, so $\Phi_k(T) \leq 4$. In both cases the second part of the lemma is true. Each increase of ℓ by two or t by one increases $\Phi_{k-1}(T)$ by two and increases the upper bound of $\lceil \ell/2 \rceil + t + 1$ on $\Phi_k(T)$ by one, which preserves the inequality $\Phi_{k-1}(T) \geq (5/4)\Phi_k(T)$. □

Lemma 4.10 gives a potential drop for any active tree T such that $\Phi_{k-1}(T) \geq 4$. To obtain a potential drop if $\Phi_{k-1}(T) < 4$, we need to consider two rounds if $\Phi_{k-1}(T) = 3$ and three rounds if $\Phi_{k-1}(T) = 2$.

Lemma 4.11. *Let T be an active tree in round $k > 2$ of RA such that $\Phi_{k-1}(T) = 3$. Then $\Phi_{k-2}(T) \geq (4/3)\Phi_k(T)$.*

Proof. The lemma holds if T has at least two active constituent trees in round $k-2$ or one with an edge end of depth two or more, since then $\Phi_{k-2}(T) \geq 4$. Suppose neither of these cases occurs. Then T has one active constituent tree, say T_2, in round $k-2$. By Lemma 4.8, no two passive constituent trees of T in round $k-2$ are connected by an edge. Thus, all passive constituent trees of T in round $k-2$ are connected by an edge with the root or a child of the root of T_2. Let T_1 be the active constituent tree of T in round $k-1$. The root of T_2 is the root or a child of the root of T_1. It follows that each passive constituent tree of T in round $k-1$ has an edge connecting its root with that of T_1. Hence, the tree containing the vertices of T_1 formed by the connect step in round k has no edge ends of depth greater than two, which implies that T has no edge ends of positive depth, making its potential two. □

Lemma 4.12. *Let T be an active tree in round $k > 3$ of RA such that $\Phi_{k-1}(T) = 2$. Then $\Phi_{k-3}(T) \geq (3/2)\Phi_k(T)$.*

Proof. The lemma holds if there are at least two active constituent trees of T in round $k-2$ or in round $k-3$, or there is a constituent tree in one of these rounds with an edge end of positive

depth: two active trees have a total potential of at least four, and an active tree with an edge end of depth at least one has a potential of at least three.

Suppose not. Let T_3, T_2, and T_1 be the active constituent trees of T in rounds $k-3$, $k-2$, and $k-1$, respectively. Since the constituent trees of T in round $k-3$ other than T_3 are passive, no edge connects any pair of them by Lemma 4.8. Since all their vertices are in T, each such tree must have an edge connecting its root with that of T_3. The connect step in round $k-2$ makes the minimum vertex in T the root of T_2. Tree T_2 has depth at most two by Lemma 3.2, since only its root can be an edge end. The connect step in round $k-1$ links each passive constituent tree of T in round $k-2$ to T_2, forming a tree S_1 containing all the vertices of T and of depth at most two. The shortcut in round $k-1$ transforms S_1 to T_1, which is flat since S_1 has depth at most two. But since T_1 is flat and contains all the vertices in T, it must be passive in round k, a contradiction. □

Having covered all the cases, we are ready to put them together. Let $a = (3/2)^{1/3} = 1.1447+$, and let $|T|$ be the number of vertices in tree T.

Lemma 4.13. *Let T be an active tree in round k of RA. Then $\Phi_k(T) \le (3/2)|T|/a^{k-3}$.*

Proof. The proof is by induction on k. Since T contains at least two vertices and has potential at most $|T|+1$, it has potential at most $(3/2)|T|$. This gives the lemma for $k \le 3$. Let $k > 3$ and suppose the lemma holds for smaller values. We consider three cases. If T contains an edge end of depth at least two, then $\Phi_k(T) \le (4/5)\Phi_{k-1}(T) \le (4/5)(3/2)|T|/a^{k-4} \le (3/2)|T|/a^{k-3}$ by Lemma 4.10, the induction hypothesis, the linearity of the total potential, and the inequality $a \le 5/4$. If the maximum depth of an edge end in T is one, then $\Phi_k(T) \le (3/4)\Phi_{k-2}(T) \le (3/4)(3/2)|T|/a^{k-5} \le (3/2)|T|/a^{k-3}$ by Lemma 4.11, the induction hypothesis, the linearity of the total potential, and the inequality $a \le (4/3)^{1/2}$. If no vertex in T other than the root is an edge end, then $\Phi_k(T) \le (2/3)\Phi_{k-3}(T) \le (2/3)(3/2)|T|/a^{k-6} \le (3/2)|T|/a^{k-3}$ by Lemma 4.12, the induction hypothesis, the linearity of the total potential, and $a = (3/2)^{1/3}$. □

Theorem 4.14. *Algorithm RA takes $O(\lg n)$ steps.*

Proof. By Lemma 4.13, if k is such that $a^{k-3} > (3/2)n$, then no tree can be active in round k. Only the last round has no active trees. □

We can use the same approach to prove an $O(\lg n)$ step bound for algorithm R, but the details are more complicated. Algorithm RA has the advantage over R that the alter step in effect does extra flattening. If T is active in round k and T_1 is the only active constituent tree of T in round $k-1$, it is possible for the depth of T_1 to be one and that of T to be two. This is not a problem in RA, because the alter decreases the depth of each edge end by one. But in R we need to give extra potential to trees of depth one to make the potential function non-increasing.

In R we define the individual potential of a tree T at the end of round k to be zero if T is passive in round k, or the depth of T if T is active in round k and not flat, or two if T is active in round k and flat. As in RA, if T exists at the end of round k and $j \le k$, we define the potential $\Phi_j(T)$ of T at the end of round j to be the sum of the potentials of its constituent trees at the end of round j, and we define the total potential at the end of round k to be the sum of the potentials of the trees existing at the end of round k. We prove analogues of Lemmas 4.10–4.13 and Theorem 4.14 for R.

Lemma 4.15. *Let T be active in round $k > 1$ of R. Then $\Phi_{k-1}(T) \ge \Phi_k(T)$, and if $\Phi_{k-1}(T) \ge 4$, then $\Phi_{k-1}(T) \ge (5/4)\Phi_k(T)$.*

Proof. Let ℓ be the sum of the depths of the active constituent trees of T in round $k-1$, let t be the number of these trees, and let f be the number of these trees that are flat. Then $\ell \ge t$ and

$\Phi_{k-1}(T) = \ell + f \geq \ell$. Let S be the tree formed from the constituent trees of T by the connect step in round k. This step in round k does not make a leaf into a non-leaf, nor does it make a root of a passive tree the parent of another such root by Lemma 4.8. It follows that any path in S contains at most $\ell + 2$ vertices: at most $\ell - t$ non-leaf vertices of active constituent trees, at most $t + 1$ roots of passive constituent trees, and at most one leaf of some constituent tree. The shortcut in round k transforms S into T, so the depth of T is at most $\lceil \ell/2 \rceil + 1$, as is its potential: if $\ell = 1$, $\lceil \ell/2 \rceil + 1 = 2$, which is the potential of a flat active tree.

If $\ell = 1$, there is one active constituent tree, and it is flat, so $\Phi_{k-1}(T) = 2 = \Phi_k(T)$, making the lemma true. If $\ell \geq 2$, $\Phi_{k-1}(T) \geq \ell \geq \lceil \ell/2 \rceil + 1 \geq \Phi_k(T)$. If $\ell = 4$, $\Phi_{k-1}(T) \geq 4$ and $\Phi_k(T) \leq 3$. If $\ell = 5$, $\Phi_{k-1}(T) \geq 5$ and $\Phi_k(T) \leq 4$. Thus, the lemma is true if $\ell \leq 5$. Each increase of ℓ by two increases the lower bound of ℓ on $\Phi_{k-1}(T)$ by two and increases the upper bound of $\lceil \ell/2 \rceil + 1$ on $\Phi_k(T)$ by one, which preserves the inequality $\Phi_{k-1}(T) \geq (5/4)\Phi_k(T)$, so the lemma holds for all ℓ by induction. □

Lemma 4.15 gives a potential drop for any active tree T such that $\Phi_{k-1}(T) \geq 4$. To obtain a potential drop if $\Phi_{k-1}(T) < 4$, we need to consider two rounds if $\Phi_{k-1}(T) = 3$ and five rounds if $\Phi_{k-1}(T) = 2$.

Lemma 4.16. *Let T be an active tree in round $k > 2$ of R such that $\Phi_{k-1}(T) = 3$. Then $\Phi_{k-2}(T) \geq (4/3)\Phi_k(T)$.*

Proof. If the constituent trees of T in round $k - 1$ or in round $k - 2$ include at least two active trees, or one active tree of depth at least four, or T has depth at most two, the lemma holds.

Suppose not. Let T_2 and T_1 be the unique active constituent trees of T in rounds $k - 2$ and $k - 1$, respectively. Let S_1 and S be the trees containing the vertices of T_2 and T_1 formed by the connect steps in rounds $k - 2$ and $k - 1$, respectively. Trees T_2, T_1, and T all have depth three, and S_1 and S have depth five. No edge connects two passive constituent trees of T in round $k - 2$, so each such tree is connected to T_2 by an edge. Call such a tree *primary* if it has an edge connecting it with the root or a child of the root of T_2, and *secondary* otherwise.

Since S_1 has depth five and T_2 has depth three, their roots must be different, so the root of S_1 is the minimum of the roots of the primary trees. Each vertex in T_2 that is the end of an edge whose other end is in a secondary tree has depth at least two in T_2, depth at least three in S_1, and depth at least two in T_1, which is formed from S_1 by the shortcut in round $k - 1$. Since T_1 has depth three and S has depth five, their roots must be different. But then the root of S must be the root of one of the secondary trees, which is impossible since the root of T_1 is not the parent of any of the edge ends connecting T_1 with the secondary trees. □

Lemma 4.17. *Let T be an active tree in round $k > 5$ of R such that $\Phi_{k-1}(T) = 2$. Then $\Phi_{k-5}(T) \geq (3/2)\Phi_k(T)$.*

Proof. If for some j between $k - 5$ and $k - 1$ inclusive the constituent trees of T in round j include at least two active trees, or one active tree of depth at least three, then $\Phi_j(T) \geq 3$, so the lemma holds.

Suppose not. Then the constituent trees of T in each round from $k - 5$ to k include exactly one active tree, of depth one or two. For j between 1 and 5 inclusive let T_j be the active constituent tree of T in round $k - j$, for j between 1 and 4 inclusive let S_j be the tree containing the vertices of T_{j+1} formed by the connect step in round $k - j$, and let S be the tree containing the vertices of T_1 formed by the connect in round k. For j from 1 to 4 inclusive, the shortcut in round $k - j$ transforms S_j into T_j, and the shortcut in round k transforms S into T.

No edge connects two passive constituent tree of T in round $k-5$, so each such tree has an edge connecting it with T_5. Call such a tree T *primary* if it has an edge connecting it with the root of T_t or to a child of the root of T_5, *secondary* otherwise. Since T_5 has depth at most two, each secondary tree has an edge connecting it with a grandchild of the root of T_5.

We consider two cases: the roots of T_5 and T_4 are the same, or they are different. In the former case, the roots of all primary trees are greater than the root of T_5, and the connect in round $k-4$ makes all of them children of the root of T_5. In the latter case, the root of T_4 is the minimum of the roots of the primary trees, and each such tree other than the one of minimum root is linked to T_5 in round $k-4$ or to T_4 in round $k-3$.

Now consider the secondary trees. If the roots of T_5 and T_4 are the same, then after the shortcut in round $k-4$ each secondary tree has an edge connecting it with the root or a child of the root of T_4. By the argument in the preceding paragraph, each such tree will be linked with T_4 in round $k-3$ or with T_3 in round $k-2$. If the roots of T_5 and T_4 are different, none of the secondary trees has an edge connecting it with the root or a child of the root of T_4 at the end of round $k-4$. In this case the roots of T_4 and T_3 must be the same, so after the shortcut in round $k-3$ each secondary tree has an edge connecting it with the root or a child of the root of T_3. Each such tree will be linked with T_3 in round $k-2$ or with T_2 in round $k-1$. Furthermore, the roots of T_2 and T_1 must be the same.

It follows that there is only one constituent tree of T in round $k-1$, and this tree is flat. But this tree must be T, making T passive in round k, a contradiction. □

Let $b = (3/2)^{1/5} = 1.0844+$.

Lemma 4.18. *Let T be an active tree in round k. Then $\Phi_k(T) \le (3/2)|T|/b^{k-5}$.*

Proof. The proof is by induction on k. Since T contains at least two vertices and has potential at most $|T|+1$, it has potential at most $(3/2)|T|$. This gives the lemma for $k \le 5$. Suppose $k > 5$ and suppose the lemma holds for smaller values. We consider three cases. If the depth of T exceeds three, then $\Phi_k(T) \le (4/5)\Phi_{k-1}(T) \le (4/5)(3/2)|T|/b^{k-6} \le (3/2)|T|/b^{k-5}$ by Lemma 4.15, the induction hypothesis, the linearity of the total potential, and the inequality $b \le 5/4$. If the depth of T is three, then $\Phi_k(T) \le (3/4)\Phi_{k-2}(T) \le (3/4)(3/2)|T|/b^{k-7} \le (3/2)|T|/b^{k-5}$ by Lemma 4.16, the induction hypothesis, the linearity of the total potential, and the inequality $b \le (4/3)^2$. If the depth of T is at most two, then $\Phi_k(T) \le (2/3)\Phi_{k-5}(T) \le (2/3)(3/2)|T|/b^{k-10} \le (3/2)|T|/b^{k-5}$ by Lemma 4.17, the induction hypothesis, the linearity of the total potential, and $b \le (3/2)^{1/5}$. □

Theorem 4.19. *Algorithm R takes $O(\lg n)$ steps.*

Proof. By Lemma 4.18, if k is such that $b^{k-5} > (3/2)n$, then no tree can be active in round k. Only the last round has no active trees. □

For the variants of R and RA that do two shortcuts in each round instead of just one, we can simplify the analysis and improve the constants. For RA, we let the potential of an active tree be the maximum depth of an edge end (or zero if there are no edge ends) plus one. The potential of an active tree drops from the previous round by at least a factor of two unless it has only one active constituent tree in the previous round and that tree has potential one. The proof of Lemma 4.12 gives a potential reduction of at least a factor of two in at most three rounds in this case. Lemma 4.13 holds with a replaced by $2^{1/3} = 1.2599+$. For R, we let the potential of an active tree be its depth. The potential of an active tree drops from the previous round by at least a factor of at least two unless it has only one active constituent tree in the previous round and that tree is flat. The proof of Lemma 4.17 gives a potential reduction of at least a factor of two in at most three rounds, and

Lemma 4.18 holds with b replaced by $2^{1/3}$, the same constant as for the two-shortcut variant of RA.

This analysis suggests that doing two shortcuts per round rather than one might improve the practical performance of R and RA, especially since a shortcut needs only n processes, but a connect step needs m. Exactly how many shortcuts to do per round is a question for experiments to resolve. Our analysis of algorithm S suggests that doing a non-constant number of shortcuts per round is likely to degrade performance. We have analyzed the one-shortcut-per-round algorithms R and RA, even though their analysis is more complicated than the corresponding two-shortcut-per-round algorithms, because this analysis applies to the corresponding algorithms with any constant number of shortcuts per round, and our goal is to determine the *simplest* algorithms with an $O(\lg n)$ step bound.

5 RELATED WORK

In this section we review previous work related to ours. We have presented our results first, since they provide insights into the related work. As far as we can tell, all our algorithms are novel and simpler than previous algorithms, although they are based on some of the previous algorithms.

Two different communities have worked on concurrent connected components algorithms, in two overlapping eras. First, theoretical computer scientists developed provably efficient algorithms for various versions of the PRAM model. This work began in the late 1970's and reached a natural conclusion in the work of Halperin and Zwick [10, 11], who gave $O(\lg n)$-step, $O(m)$-work randomized algorithms for the **EREW (exclusive read, exclusive write)** PRAM. Their second algorithm finds spanning trees of the components. The EREW PRAM is the weakest variant of the PRAM model, and finding connected components in this model requires $\Omega(\lg n)$ steps [7]. To solve the problem sequentially takes $O(m)$ time, so the Halperin-Zwick algorithms minimize both the number of steps and the total work (number of steps times the number of processes). Whether there is a deterministic EREW PRAM algorithm with the same efficiency remains an open problem.

Halperin and Zwick's paper [11] contains a table listing results preceding theirs, and we refer the reader to their paper for these results. Our interest is in simple algorithms for a more powerful computational model, so we content ourselves here with discussing simple labeling algorithms related to ours. (The Halperin-Zwick algorithms and many of the preceding ones are *not* simple.) First, we review variants of the PRAM model and how they relate to our algorithmic framework.

The three main variants of the PRAM model, in increasing order of strength, are EREW, **CREW (concurrent read, exclusive write)**, and **CRCW (concurrent read, concurrent write)**. The CRCW PRAM has four standard versions that differ in how they handle write conflicts: (i) COMMON: all writes to the same location at the same time must be of the same value; (ii) ARBITRARY: among concurrent writes to the same location, an arbitrary one succeeds; (iii) PRIORITY: among concurrent writes to the same location, the one done by the highest-priority process succeeds; (iv) COMBINING: values written concurrently to a given location are combined using some symmetric function. As discussed in Section 2, our algorithms can be implemented on a COMBINING CRCW PRAM, with minimization as the combining function.

An early and important theoretical result is the $O(\lg^2 n)$-step CREW PRAM algorithm of Hirschberg, Chandra, and Sarwate [12]. Algorithm S is a simplification of their algorithm. They represent the graph by an adjacency matrix, but it is easy to translate their basic algorithm into our framework. Their algorithm alternates connect steps with repeated shortcuts. To do connection, they use a variant of *parent-connect* that we call *strong-parent-connect*. It concurrently sets $x.p$ for each vertex x equal to the minimum $w.p \neq x$ such that there is an edge $\{v, w\}$ with $v.p = x$; if there is no such edge, $x.p$ does not change. The following pseudocode implements this method in our framework:

strong-parent-connect:
  for each vertex v **do**
    $v.n = \infty$
  for each edge $\{v, w\}$ **do**
    if $v.p \neq w.p$ **then**
      $v.p.n = \min\{v.p.n, w.p\}$
      $w.p.n = \min\{w.p.n, v.p\}$
  for each vertex v **do**
    if $v.n \neq \infty$ **then**
      $v.p = v.n$

This version of connection can make a larger vertex the parent of a smaller one. Thus, their algorithm does not do minimum labeling. Furthermore it can create parent cycles of length two, which Hirschberg et al. eliminate in a cleanup step at the end of each round. To do the cleanup it suffices to concurrently set $v.p = v$ for each vertex such that $v.p > v$ and $v.p.p = v$. Their algorithm is one of two we have found in the literature that can create parent cycles.

Although they do not say this, we think the reason Hirschberg et al. used *strong-parent-connect* was to guarantee that each tree links with another tree in each round. This gives them an $O(\lg n)$ bound on the number of rounds and an $O(\lg^2 n)$ bound on the number of steps, since there are $O(\lg n)$ shortcuts per round. Our simpler algorithm S uses *parent-connect* in place of *strong-parent-connect*, making it a minimum labeling algorithm and eliminating the cleanup step. Although *parent-connect* does not guarantee that each tree links with another tree every round, it does guarantee such linking every two rounds, giving us the same $O(\lg^2 n)$ step bound as Hirschberg et al. See the proof of Theorem 4.1.

The first $O(\lg n)$-step PRAM algorithm was that of Shiloach and Vishkin [23]. It runs on an ARBITRARY CRCW PRAM, as do the other algorithms we discuss, except as noted. The following is a version of their algorithm SV in our framework:

Algorithm SV:
  repeat
    {*shortcut*; *arb-parent-root-connect*; *stagnant-parent-root-connect*; *shortcut*}
  until no $v.p$ changes

arb-parent-root-connect:
  for each vertex v **do**
    $v.o = v.p$
  for each edge $\{v, w\}$ **do**
    if $v.o > w.o$ and $v.o = v.o.o$ **then**
      $v.o.p = w.o$
    else if $w.o = w.o.o$ **then**
      $w.o.p = v.o$

and *stagnant-parent-root-connect* is the following variant:

stagnant-parent-root-connect:
  for each vertex v **do**
    $v.o = v.p$
  for each edge $\{v, w\}$ **do**
    if $v.o \neq w.o$ **then**

if $v.o$ is a stagnant root **then**
  $v.o.p = w.o$
else if $w.o$ is a stagnant root **then**
  $w.o.p = v.o$

Whereas *parent-root-connect* updates the parent of each root x to be the *minimum* y such that there is an edge $\{v, w\}$ with $x = v.o$ and $y = w.o < v.o$ if there is such an edge, *arb-parent-root-connect* replaces the parent of each such root by an *arbitrary* such y. Arbitrary resolution of write conflicts suffices to implement the latter method, but not the former.

Shiloach and Vishkin define a root to be *stagnant* if its tree is not changed by the first two steps of the main loop (the first shortcut and the *arb-parent-root-connect*). Their algorithm has additional steps to keep track of stagnant roots. Method *stagnant-parent-root-connect* updates the parent of each stagnant root x to be an arbitrary y such that there is an edge $\{v, w\}$ with $x = v.o$ and $y = w.o \neq v.o$ if there is such an edge. The definition of "stagnant" implies that no two stagnant trees are connected by an edge.

Algorithm SV does not do minimum labeling, since *stagnant-parent-root-connect* can make a larger vertex the parent of a smaller one. Nevertheless, the algorithm creates no cycles, although the proof of this is not straightforward, nor is the efficiency analysis.

Algorithm R is algorithm SV with the third and fourth steps of the main loop deleted and the second step modified to resolve concurrent writes by minimum value instead of arbitrarily. Shiloach and Vishkin state that one shortcut can be deleted from their algorithm without affecting its asymptotic efficiency. They included the third step for two reasons: (i) their analysis examines one round at a time, requiring that every tree change in every round, and (ii) if the third step is deleted, the algorithm can take $\Omega(n)$ steps on a graph that is a tree with edges $\{i, n\}$ for $i \in [n-1]$. This example strongly suggests that to obtain a simpler algorithm one needs to use a more powerful model of computation, as we have done by using minimization to resolve write conflicts.

Awerbuch and Shiloach presented a slightly simpler $O(\lg n)$-step algorithm and gave a simpler efficiency analysis [3]. Our analysis of algorithms R and RA in Section 4.3 uses a variant of their potential function. Their algorithm is algorithm SV with the first shortcut deleted and the two connect steps modified to update only parents of roots of flat trees. The computation needed to keep track of flat tree roots is simpler than that needed in algorithm SV to keep track of stagnant roots.

An even simpler but randomized $O(\lg n)$-step algorithm was proposed by Reif [22]:

Algorithm Reif:
  repeat
    {for each vertex flip a coin; *random-parent-connect*; *shortcut*}
  until no $v.p$ changes

where *random-parent-connect* is:

random-parent-connect:
  for each vertex v **do**
    $v.o = v.p$
  for each edge $\{v, w\}$ **do**
    if $v.o$ flipped heads and $w.o$ flipped tails **then**
      $v.o.p = w.o$
    else if $w.o$ flipped heads and $v.o$ flipped tails **then**
      $w.o.p = v.o$

Reif's algorithm keeps the trees flat, making the algorithm monotone, although it does not do minimum labeling. Although it is randomized, Reif's algorithm is simpler than those of Shiloach and Vishkin, but R and RA are even simpler and are deterministic.

We know of one algorithm other than that of Hirschberg et al. [12] that does not maintain acyclicity. This is the algorithm of Johnson and Metaxis [16]. Their algorithm runs in $O((\lg n)^{3/2})$ steps on an EREW PRAM. It uses a form of shortcutting to eliminate any cycles created by connection steps.

Algorithms that run on a more restricted form of PRAM, or use fewer processes (and thereby do less work) use various kinds of edge alteration, edge addition, and edge deletion, along with techniques to resolve read and write conflicts. Such algorithms are much more complicated than those we have considered. Again, we refer the reader to [10, 11] for results and references.

The second era of concurrent connected components algorithms was that of the experimentalists. It began in the 1990's and continues to the present. Experimentation has expanded greatly with the growing importance of huge graphs representing the internet, the world-wide web, friendship connections, and other symmetric relations, as well as the development of cloud computing frameworks. These trends make concurrent algorithms for connected components both practical and useful. The general approach of the experimentalists has been to take one or more existing algorithms, possibly simplify or modify them, implement the resulting suite of algorithms on one or more computing platforms, and report the results of experiments done on some collection of graphs. Examples of such studies include [8, 9, 13, 14, 20, 26, 29, 30].

Some of these papers make claims about the theoretical efficiency of algorithms they propose, but several of these claims are incorrect or unjustified. We give some examples. The first is a paper by Greiner [9] in which he claims an $O(\lg^2 n)$ step bound for his "hybrid" algorithm.

Greiner's description of this algorithm is incomplete. The algorithm is a modification of the algorithm of Hirschberg et al. [12]. Each round does a form of direct connect followed by repeated shortcuts followed by an alteration. Since repeated shortcuts guarantee that all trees are flat at the beginning of each round, this is equivalent to using a version of *parent-connect* and not doing alteration. The main novelty in his algorithm is that alternate rounds use maximization instead of minimization in the connect step. He does not specify exactly how the connect step works. There are at least two possibilities. One is to use *direct-connect*, but in alternate rounds replace min by max. The resulting algorithm is a min-max version of algorithm S.

The second possibility is to use the following strong version of *direct-connect*, but in alternate rounds replace min by max and ∞ by $-\infty$:

strong-direct-connect:
  for each vertex v **do**
    $v.n = \infty$
  for each edge $\{v, w\}$ **do**
    $v.n = \min\{v.n, w\}$
    $w.n = \min\{w.n, v\}$
  for each vertex v **do**
    if $v.n \neq \infty$ **then**
      $v.p = v$

The resulting algorithm is a min-max version of the Hirschberg et al. algorithm. Greiner claims an $O(\lg n)$ bound on the number of rounds and an $O(\lg^2 n)$ bound on the number of steps. But these bounds do not hold for the algorithm that uses the min-max version of *direct-connect*: on the bad example of Shiloach and Vishkin consisting of an unrooted tree with vertex n adjacent to vertices 1 through $n - 1$, the algorithm takes $\Omega(n)$ steps. This example has a high-degree vertex,

but there is a simple example whose vertices are of degree at most three, consisting of a path of odd vertices $1, 3, 5, \ldots, n$ with each even vertex i adjacent to $i + 1$.

On the other hand, the algorithm that uses the min-max version of *strong-direct-connect* can create parent cycles of length two, which must be eliminated by a cleanup as in the Hirschberg et al. algorithm. Greiner says nothing about eliminating cycles. We conclude that either his step bound is incorrect or his algorithm is incorrect.

At least one other work reproduces Greiner's error: Soman et al. [24, 25] propose a modification of Greiner's algorithm intended for implementation on a GPU model. Their algorithm is the inefficient version of Greiner's algorithm, modified to use *parent-connect* instead of *direct-connect* and without alteration.

Their specific implementation of the connect step is as follows:

alternate-connect:
  for each edge $\{v, w\}$ **do**
    if $v.p \neq w.p$ **then**
      $x = \min\{v.p, w.p\}$
      $y = \max\{v.p, w.p\}$
      if round is even **then**
        $y.p = x$
      else $x.p = y$

Soman et al. say nothing about how to resolve concurrent writes. If this resolution is arbitrary, or by minimum in the even rounds and by maximum in the odd rounds, then the algorithm takes $\Omega(n)$ steps on the examples mentioned above.

Algorithm S, and the equivalent algorithm that uses *direct-connect* and alteration, are simpler than the algorithms of Greiner and Soman et al. and have guaranteed $O(\lg^2 n)$ step bounds. We conclude that alternating minimization and maximization adds complication without improving efficiency, at least in theory.

Another paper that has an invalid efficiency bound as a result of not handling concurrent writes carefully is that of Yan et al. [29]. They consider algorithms in the PREGEL framework [19], which is a graph-processing platform designed on top of the MPC model. All the algorithms they consider can be expressed in our framework. They give an algorithm obtained from algorithm SV by deleting the first shortcut and replacing the second connect step by the first connect step of Awerbuch and Shiloach's algorithm. In fact, the second connect step does nothing, since any parent update it would do has already been done by the first connect step. That is, this algorithm is equivalent to algorithm SV with the first shortcut and the second connect step deleted. Their termination condition, that all trees are flat, is incorrect, since there could be two or more flat trees in the same component. They claim an $O(\lg n)$ bound on steps, but since they assume arbitrary resolution of write conflicts, the actual step bound is $\Theta(n)$ by the example of Shiloach and Vishkin.

A third paper with an analysis gap is that of Stergio, Rughwani, and Tsioutsiouliklis [26]. They present an algorithm that we call SRT, whose main loop expressed in our framework is the following:

Algorithm SRT:
  repeat
    for each vertex v **do**
      $v.o = v.p$
      $v.n = v.p$
    for each edge $\{v, w\}$ **do**

```
      if v.o > w.o then
        v.n = min{v.n, w.o}
      else w.n = min{w.n, v.o}
    for each vertex v do
      v.o.p = min{v.o.p, v.n}
    for each vertex v do
      v.p = min{v.p, v.n.o}
  until no v.p changes
```

This algorithm does an extended form of connection combined with a variant of shortcutting that combines old and new parents. It is not monotone. Stergio et al. implemented this algorithm on the Hronos computing platform and successfully solved problems with trillions of edges. They claimed an $O(\lg n)$ step bound for the algorithm, but we are unable to make sense of their analysis. Their paper motivated our work.

A recently proposed algorithm similar to SRT is FastSV, proposed by Zhang, Azad, and Hu [30]. This algorithm is not monotone. It combines connecting and shortcutting. In each round it computes the grandparent $v.g = v.p.p$ of each vertex v and uses $v.g$ as a candidate for $v.p$, $w.p.p$ and $w.p$, for each edge $\{v, w\}$. They did not analyze FastSV but instead compared it experimentally to a simplified version of Awerbuch and Shiloach's algorithm called LACC and to other variants of SV. In their experiments SV was fastest.

We have been unable to prove a worst-case polylogarithmic step bound for SRT, nor for FastSV, nor indeed for any non-monotone algorithm that does not use edge alteration. But we do not have bad examples for these algorithms either.

A final paper with an interesting algorithm but incorrect analysis is that of Burkhardt [6]. The main novelty in Burkhardt's algorithm is to replace each edge $\{v, w\}$ by a pair of oppositely directed arcs (v, w) and (w, v) and to use *asymmetric* alteration: he replaces (v, w) by $(w, v.p)$ instead of $(v.p, w.p)$ (unless $w = v.p$). This idea allows him to combine connecting and shortcutting in a natural way. (Burkhardt claims that his algorithm does not do shortcutting, but it does, implicitly.) Burkhardt does not give an explicit stopping rule, saying only, "This is repeated until all labels converge." An iteration can alter arcs without changing any parents, so one must specify the stopping rule carefully. The following is a version of the main loop of Burkhardt's algorithm with the parent updates and the arc alterations disentangled, and which stops when there is one root and all other vertices are leaves:

```
Algorithm B:
  repeat
    for each arc (v, w) do
      if v > w then
        v.p = min{v.p, w}
    for each arc (v, w) do
      if v.p ≠ w then
        replace (v, w) by (w, v.p)
      else delete (v, w)
    for each vertex v do
      if v.p ≠ v then
        add arc (v.p, v)
  until every arc (v, w) has v.p = w.p and v.p ∈ {v, w}
```

Burkhardt claims that his algorithm takes $O(\lg d)$ steps, which would be remarkable if true. Unfortunately, a long skinny grid is a counterexample, as shown in [2]. Burkhardt also claimed

that the number of arcs existing at any given time is at most $2m + n$. The version above has a $2m$ upper bound on the number of arcs. Two small changes in the algorithm reduce the upper bound on the number of arcs to m and make the shortcutting more efficient: replace each original edge $\{v, w\}$ by *one* arc $(\max\{v, w\}, \min\{v, w\})$, and in the loop over the vertices replace "add arc $(v.p, v)$" by "add arc $(v, v.p.p)$." We call the resulting algorithm AA, for *asymmetric alteration*.

The following pseudocode implements this algorithm:

```
Algorithm AA:
  for each vertex v do v.p = v
  for each edge {v, w} do
    replace {v, w} by arc (max{v, w}, min{v, w})
  repeat
    for each arc (v, w) do
      v.p = min{v.p, w}
    for each arc (v, w) do
      delete (v, w)
      if w ≠ v.p then
        add arc (w, v.p)
    for each vertex v do
      if v ≠ v.p then
        add arc (v, w.p)
  until no arc (v, w) has w ≠ v.p
```

A version of Algorithm AA was proposed to us by Yu-Pei Duo [private communication, 2018]. The techniques of Section 4.2 extend to give an $O(\lg^2 n)$ step bound for AA, B, and Burkhardt's original algorithm. We omit the details.

Very recently, theoreticians have become interested in concurrent algorithms for connected components again, with the aim of obtaining a step bound logarithmic in d rather than n, for a suitably powerful model of computation. The first breakthrough result in this direction was that of Andoni et al. [2]. They gave a randomized algorithm that takes $O(\lg d \lg \log_{m/n} n)$ steps in the MPC model. Their algorithm uses graph densification based on the distance-doubling technique of [21], controlled to keep the number of edges linearly bounded. Behnezhad et al. [5] improved the result of Andoni et al. by reducing the number of steps to $O(\lg d + \lg \log_{m/n} n)$. Their algorithm can be implemented in the MPC model or on a very powerful version of the CRCW PRAM that supports a "multiprefix" operation. In recent work [18], we show that this algorithm and that of Andoni et al. can be simplified and implemented on an ARBITRARY CRCW PRAM.

6 REMARKS

We have presented several very simple label-update algorithms to compute connected components concurrently. Our best bounds, of $O(\lg n)$ steps and $O(m \lg n)$ work, are for two related monotone algorithms, R and RA. For two other algorithms, A, which is non-monotone, and S, which keeps all trees flat by doing repeated shortcuts, our bounds are $O(\lg^2 n)$ steps and $O(m \lg^2 n)$ work, which are tight for S but maybe not for A. We have also pointed out errors in previous analyses of similar algorithms.

Our analysis of these algorithms is novel in that it extends over several rounds of the main loop, unlike previous analyses that consider only one round at a time. Our analysis of A combines new ideas with an idea from the analysis of disjoint set union algorithms. As mentioned in Section 5, this analysis extends to give an analysis of another algorithm in the literature (B) and to a variant of this algorithm (AA).

Our results illustrate the subtleties of even simple algorithms. A number of theoretical questions remain open, notably, determining tight asymptotic step bounds for algorithms P, A, H, SRT, FastSV, AA, and B. For P, SRT, and FastSV we know nothing interesting: our techniques seem too weak to derive a poly-logarithmic step bound for an algorithm such as these that is non-monotone and in which trees can be passive for an indefinite number of rounds. For A, AA, and B we have a bound of $O(\lg^2 n)$ steps but the lower bound is $\Omega(\lg n)$.

All our algorithms are simple enough to merit experiments. Indeed, at least one recent experimental study using GPUs [13] has included our algorithms. In this study, our algorithms were faster than SV and much faster than simple label propagation (no shortcuts), but algorithms using disjoint set union were faster, and random sampling of the graph edges sped up all algorithms.

There is a natural way to convert each of the deterministic algorithms we have considered into a randomized algorithm: number the vertices from 1 to n uniformly at random and identify the vertices by number. In an application, one may get such randomization for free, for example if a hash table stores the vertex identifiers. It is natural to study the efficiency that results from such randomization. Working with Eitan Zlatin, we have obtained a high-probability $O(\lg n)$ or $O(\lg^2 n)$ step bound for the randomized version of most of the algorithms we have presented, and in particular an $O(\lg n)$ bound for algorithm A, improving our $O(\lg^2 n)$ worst-case bound. We shall report on these results in the future.

We have assumed global synchronization. The problem becomes much more challenging in an asynchronous setting. One of the authors and a colleague have developed algorithms for asynchronous concurrent disjoint set union [15], the incremental version of the connected components problem. Their algorithms can be used to find connected components asynchronously.

An interesting extension of the connected components problem is to construct a spanning tree of each component. It is easy to extend algorithms R, RA, and S to do this: when an edge causes a root to become a child, add the corresponding original edge to the spanning forest. Extending non-monotone algorithms such as A to construct spanning trees seems a much bigger challenge.

ACKNOWLEDGMENTS

We thank Dipen Rughwani, Kostas Tsioutsiouliklis, and Yunhong Zhou for telling us about [26], for extensive discussions about the problem and our algorithms, and for insightful comments on our early results.

In the preliminary version of this work [17], we claimed an $O(\lg^2 n)$ step bound for algorithm P and for a related algorithm E (which we have omitted from the current paper). We thank Pei-Duo Yu for discovering that Lemma 6 in [17] does not hold for P and E, invalidating our proof of Theorem 13 in [17] for these algorithms.

REFERENCES

[1] Selim G. Akl. 1989. *Design and Analysis of Parallel Algorithms*. Prentice Hall.

[2] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. 2018. Parallel graph connectivity in log diameter rounds. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7–9, 2018*. 674–685.

[3] Baruch Awerbuch and Yossi Shiloach. 1987. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Computers* 36, 10 (1987), 1258–1263.

[4] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication steps for parallel query processing. *J. ACM* 64, 6 (2017), 40:1–40:58.

[5] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab S. Mirrokni. 2019. Near-optimal massively parallel graph connectivity. *CoRR* abs/1910.05385 (2019).

[6] Paul Burkhardt. 2018. Graph connectivity in log-diameter steps using label propagation. *CoRR* abs/1808.06705 (2018).

[7] Stephen A. Cook, Cynthia Dwork, and Rüdiger Reischuk. 1986. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.* 15, 1 (1986), 87–97.

[8] Steve Goddard, Subodh Kumar, and Jan F. Prins. 1994. Connected components algorithms for mesh-connected parallel computers. In *Parallel Algorithms, Proceedings of a DIMACS Workshop, Brunswick, New Jersey, USA, October 17–18, 1994*. 43–58.
[9] John Greiner. 1994. A comparison of parallel algorithms for connected components. In *SPAA*. 16–25.
[10] Shay Halperin and Uri Zwick. 1996. An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.* 53, 3 (1996), 395–416.
[11] Shay Halperin and Uri Zwick. 2001. Optimal randomized EREW PRAM algorithms for finding spanning forests. *Journal of Algorithms* 39, 1 (2001), 1–46.
[12] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. 1979. Computing connected components on parallel computers. *Commun. ACM* 22, 8 (1979), 461–464.
[13] Changwan Hong, Laxman Dhulipala, and Julian Shun. 2020. Exploring the design space of static and incremental graph connectivity algorithms on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 55–69.
[14] Tsan-Sheng Hsu, Vijaya Ramachandran, and Nathaniel Dean. 1997. Parallel implementation of algorithms for finding connected components in graphs. *Parallel Algorithms: Third DIMACS Implementation Challenge, October 17–19, 1994* 30 (1997), 20.
[15] Siddhartha V. Jayanti and Robert E. Tarjan. 2016. A randomized concurrent algorithm for disjoint set union. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25–28, 2016*. 75–82.
[16] Donald B. Johnson and Panagiotis Takis Metaxas. 1997. Connected components in O (logˆ3/2 n) parallel time for the CREW PRAM. *J. Comput. Syst. Sci.* 54, 2 (1997), 227–242.
[17] S. Cliff Liu and Robert E. Tarjan. 2019. Simple concurrent labeling algorithms for connected components. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019, January 8–9, 2019 - San Diego, CA, USA*. 3:1–3:20.
[18] S. Cliff Liu, Robert E. Tarjan, and Peilin Zhong. 2020. Connected components on a PRAM in log diameter time. *CoRR* abs/2003.00614 (2020). https://arxiv.org/abs/2003.00614.
[19] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6–10, 2010*. 135–146.
[20] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18–20, 2015*.
[21] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. 2013. Finding connected components in map-reduce in logarithmic rounds. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8–12, 2013*. 50–61.
[22] John H. Reif. 1984. *Optimal Parallel Algorithms for Graph Connectivity*. Technical Report. Harvard University Cambridge, MA, Aiken Computation Lab.
[23] Yossi Shiloach and Uzi Vishkin. 1982. An O(log n) parallel connectivity algorithm. *J. Algorithms* 3, 1 (1982), 57–67.
[24] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19–23 April 2010 - Workshop Proceedings*. 1–8.
[25] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. 2010. Some GPU algorithms for graph connected components and spanning tree. *Parallel Processing Letters* 20, 4 (2010), 325–339. https://doi.org/10.1142/S0129626410000272
[26] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5–9, 2018*. 540–546.
[27] Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case analysis of set union algorithms. *J. ACM* 31, 2 (1984), 245–281.
[28] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.
[29] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB* 7, 14 (2014), 1821–1832.
[30] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. 2020. FastSV: A distributed-memory connected component algorithm with fast convergence. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 46–57.

Received March 2020; revised November 2021; accepted June 2022