Finding minimum spanning forests in logarithmic time and linear work using random sampling

Richard Cole* New York University

Philip N. Klein† Brown University

Robert E. Tarjan‡ Princeton University and NEC Research Institute

Abstract

We describe a randomized CRCW PRAM algorithm that finds a minimum spanning forest of an n-vertex graph in $O(\log n)$ time and linear work. This shaves a factor of $2^{\log^* n}$ off the best previous running time for a linear-work algorithm. The novelty in our approach is to divide the computation into two phases, the first of which finds only a partial solution. This idea has been used previously in parallel connected components algorithms.

1 Introduction

We describe the first work-optimal minimum spanning forest (MSF) algorithm that runs in $O(\log n)$ time. The algorithm uses a random-sampling technique previously used by Karger, Klein, and Tarjan in a sequential linear-time algorithm and by Cole, Klein, and Tarjan in a parallel algorithm.

These previous algorithms have the following form. Choose a random subset of edges, and recursively calculate the MSF of the *sample graph*, the graph consisting of the chosen edges. Use the recursively calculated minimum spanning forest to identify edges of the original graph that are guaranteed not to belong to the MSF. Discard these edges, and recursively calculate the MSF of the remaining graph.

Identifying the edges to be discarded seems to require $\Theta(\log n)$ time; thus the time required by an algorithm having the above form is $O(\log n)$ times the number of recursive invocations. Previously [5], by varying the sampling probability depending on the recursion depth, we were able to bound the number of invocations by $O(2^{\log^* n})$, but there seems no way to reduce it further to a constant, which is what would be necessary to achieve $O(\log n)$ time using this approach.

In this paper we resolve this dilemma by breaking the computation into two phases. In the first phase, the number of recursive invocations remains $O(2^{\log^* n})$, but we manage to reduce the time per invocation to something significantly less than $\Theta(\log n)$. The result of the first phase, however, is not the entire MSF but only a subset of it. We then contract the edges in this subset, and, in the second phase, calculate the MSF of the contracted graph. The MSF of the original graph is the MSF of the contracted graph together with the contracted edges. Because the contracted graph is significantly smaller than the input graph, we can ensure that the second phase consists of only a constant number of invocations (each taking $O(\log n)$ time). Thus the overall time bound for the two phases is $O(\log n)$.

The first phase is the challenge. How can we reduce the time per invocation? We modify the criterion by which edges can be discarded. The intuition is as follows. In the first recursive call (which operates on the sample graph), why should we completely solve the MSF problem when the solution is only intended to help us discard edges? We formulate a less ambitious goal—partial solution of the MSF problem—and show that using a partial solution for the sample graph we can quickly identify edges of the original graph that do not belong to the partial solution for the original graph. We later define what we mean by "partial solution." For the moment, we remark merely that it is a "large enough" set of edges that is a subset of the MSF.

A technical difficulty arises in carrying out the modified sampling approach. In the discarding step, we might discard edges that belong to the MSF. For this reason, if we were not careful in carrying out the second recursive call, we might include in the partial solution edges that do not belong to the MSF ("bad edges"). We therefore do not entirely discard edges but rather designate them as "out-edges" for this recursive call. The out-edges are used to guard against the partial solution getting too big and including bad edges. We show that the choice of out-edges ensures essentially that if the partial solution is about to include an out-edge, it is already big enough. We also show that the set of out-edges can be represented compactly, which is necessary for the efficiency of our algorithm.

1.1 Previous work

Several researchers have addressed the problem of giving a work-efficient parallel algorithm for finding a minimum

*This research was supported in part by NSF grants CCR-8906949 and CCR-9202900.

†This research was supported in part by NSF PYI award CCR-9157620, together with PYI matching funds from Thinking Machines Corporation, Xerox Corporation, and Honeywell Corporation. Additional support provided by ARPA contract N00014-95-1-1246.

‡Department of Computer Science, Princeton University, Princeton, NJ and the NEC Research Institute, Princeton, NJ. Research at Princeton University partially supported by the National Science Foundation, Grant No. CCR-8920505; the Office of Naval Research, Contract No. N0014-91-J-1463; and ARPA Contract N00013-95-1-1246.

spanning tree. Chin, Lam, and Chen [3] gave an algorithm that runs in $O(\log^2 n)$ time using $n^2/\log^2 n$ processors. Thus their algorithm achieves linear speed-up when the input graph is a complete graph. However, it is not very work-efficient for sparse input graphs. Awerbuch and Shiloach [1] proposed a parallel algorithm for finding a minimum spanning tree; their algorithm requires $O(\log n)$ time using $m+n$ processors, where n and m are, respectively, the number of vertices and edges of the graph. However, their result assumes a model in which write-conflicts are resolved by priority, where the priority of a processor is determined by the weight of the edge assigned to it.

Cole and Vishkin [6] have claimed an algorithm running on a CRCW PRAM that requires $O(\log n)$ time and $O((n+m)\log\log\log n/\log n)$ processors. Their algorithm assumes the same strong model as the algorithm of Awerbuch and Shiloach.

Karger [17] has claimed an algorithm running on an EREW PRAM that requires $O(\log n)$ time and $m/\log n + n^{1+\epsilon}$ processors for any constant $\epsilon > 0$. Thus his algorithm is within a constant factor of optimal for sufficiently dense graphs, and is within a fractional polynomial factor for very sparse graphs.

A related but simpler problem is that of finding connected components. Gazit [10] discovered a randomized logarithmic-time, linear-work CRCW PRAM connected-components algorithm. Halperin and Zwick [12] discovered how to test connectivity on a CREW PRAM in the same bounds using Gazit's approach, and later refined their algorithm to actually find connected components [13]. Gazit's approach was the inspiration for our two-phase MSF algorithm: Gazit's algorithm first builds pieces of components and then combines them in a second phase. Extending this idea to the MSF problem is not straightforward and is where the main technical contributions of our paper lie.

2 The top-level algorithm

We now give the top-level algorithm. It uses *contraction* of edges. Contraction of an edge with endpoints u and v results in removal of the edge and coalescing of the endpoints u and v to form a new vertex. Every other edge that had as endpoint either u or v has the new vertex as endpoint after the contraction. Thus every edge that existed in the graph before the contraction exists in the graph after contraction (except for the edge contracted), albeit with new endpoints.

> Let G_0 denote the input graph, and let m be the number of edges in G_0.
>
> Let $k := \sqrt{\log\log m}$. Note: k is a parameter used in the procedure FindForest.
>
> Call FindForest($G_0, \log^* m$).
>
> Let G_1 be the graph obtained from G_0 by contracting all edges designated as in-edges by FindForest.
>
> Call FinishUp(G_1).
>
> Output the union of the in-edges of G_1 and the in-edges of G_0.

The algorithm uses two procedures, FindForest and FinishUp. The call FindForest($G_0, \log^* m$) designates as *in-edges* a subset of the edges of G_0's minimum spanning forest (MSF). Since G_1 is obtained by contracting these edges, it follows that the MSF of G_0 consists of these edges together with the MSF of G_1. The call FinishUp(G_1) designates as in-edges all the edges of G's MSF. Thus the set of edges output by the algorithm is the MSF of G_0.

Now we preview the analysis. We show below that the call FindForest($G_0, \log^* m$) results in $O(2^{\log^* m})$ recursive invocations. Assume for simplicity that G_0 is connected. Each invocation requires expected time $O(\log m \log\log\log m/\log\log m)$, so the total time is $o(\log m)$. We show that the total work is $O(m)$. We show also that in the forest of edges of G_0 designated as in-edges, each tree has at least k edges. Hence the number of vertices in G_1 is at most $1/k$ times the number of vertices in G_0, and hence at most $1/k$ times m. The procedure FinishUp uses this fact to find the MSF of G_1 in $O(\log m)$ time using linear work.

2.1 In-edges

One correctness condition for the algorithm is the *in-edge soundness condition:*

> all in-edges belong to the MSF.

To maintain this property, the algorithm uses the following simple proposition. If all present in-edges belong to the MSF, and an edge e is the cheapest edge incident to a component of in-edges, then e is in the MSF (and can therefore safely be designated an in-edge).

The algorithm keeps track of the components of in-edges, which are trees, and the edges between these trees. When an edge is newly designated an in-edge, the trees it connects are merged into one.

2.2 Min trees

In order to precisely define the correctness condition for FindForest, we introduce some terminology about partial solutions to the MSF problem.

Recall the following algorithm[1] for finding a minimum spanning tree in a connected graph G. Initialize the set S of spanned vertices to be $\{v\}$ for some vertex v. Then repeat the following step until S contains all the vertices of G: select the cheapest edge vw incident to S, and add to S whichever endpoint is not already in S. The set of edges vw selected by the above algorithm is the minimum spanning forest of G.

If k is smaller than the number of vertices in the graph, the first k edges selected by the above algorithm form a tree containing u. We call this tree the *k-min tree of u (in G)*. The k^{th} edge chosen by this algorithm is called u's k^{th} min edge. If the connected component of G that contains u consists of fewer than $k+1$ vertices, we define u's k-Prim tree to be the minimum spanning tree of that component.

Lemma 1 *Suppose an edge e is incident to a vertex u in G, and u's k-min tree does not contain e. Then e is costlier than any edge in this tree.*

The goal of FindForest(G, i) is to identify a subset F of the MSF of G obeying the following *completeness condition:*

> for each vertex v of G, the k-min tree of v in G is contained in F.

[1] While commonly attributed to Prim, this algorithm appears in papers by Jarnik [15] and Dijkstra.

It follows from the completeness condition that the forest of in-edges of G_0 designated by the top-level call, FINDFOREST($G_0, \log^* m$), consists of trees each of size at least k, as required.

2.3 Out-edges

The efficiency of FINDFOREST is based on identifying edges that are *not* necessary for completeness; the procedure designates these edges as *out-edges*. Thus the *out-edge soundness condition* is:

> each out-edge of G does not belong to the k-mintree of any vertex in G.

Note that an edge may belong to the MSF but not to any vertex's k-min tree; hence even some MSF edges might get designated as out-edges. An edge that has not been designated an in-edge or an out-edge is said to be *neutral*.

One might think that out-edges could simply be deleted. However, since they might belong to the MSF, deleting them would cause FINDFOREST to misidentify some remaining edges as MSF edges; the procedure would determine that some edge was the cheapest neutral edge incident to a tree of in-edges, and would infer that this edge belonged to the MSF–though the cheapest edge incident to that tree might be an out-edge. To prevent such misidentification, FINDFOREST represents out-edges as follows: for each tree of in-edges, the procedure keeps track of the cheapest incident out-edge (if any). This representation can easily be updated when two such trees merge. There is no need for explicit representation of out-edges.

2.4 Borůvka steps

One basic operation used repeatedly by our algorithm is a *Borůvka step*, which we adapt from Borůvka's algorithm for finding a MSF'[2]. Let G be a graph in which some in-edges and out-edges have been designated. Call a tree of in-edges *inactive* if its cheapest incident edge has previously been determined to be an out-edge (and *active* otherwise. In our version of a Borůvka step, a subset $T_1, \ldots, T_t$ of active in-edge trees are selected, and for each tree T_i the cheapest neutral edge e_i is determined. If e_i is cheaper than the cheapest out-edge incident to T_i then e_i is designated an in-edge. Note that these edges belong to the MSF, so this step preserves in-edge soundness. We require our Borůvka step to satisfy two properties:

noninterference: For each tree T_i, the endpoint of e_i not in T_i must not belong to another selected tree T_j.

likelihood: For each tree T of in-edges, if T has an incident edge then the probability is at least $1/8$ that T is one of the trees selected.

Such a Borůvka step can be implemented to run in $O(1)$ expected time using a number of processors equal to the number of active in-edge trees plus the number of neutral edges incident to such trees [5].

3 The Phase I Algorithm: Overview

We now give the recursive procedure FINDFOREST. It refers to m, which denotes the number of edges in the original input graph. It also refers to a global parameter k whose value is $\sqrt{\log\log m}$. We use $\log^{(i)} m$ to denote the application to m of the i-fold composition of log with itself, e.g. $\log\log m$ can be written $\log^{(2)} m$. The procedure makes use of two constants a and b such that $a \geq 2b$, and a few assorted constants hidden by the big-Oh notation. The second argument to FINDFOREST is a descending counter that indicates the depth of the recursion.

FINDFOREST(G, i)

Step 0: If $i = 2$ then call BASECASE(G) and return.

Step 1: Perform $O(\log[(\log^{(i-1)} m)^a])$ Borůvka steps.

Step 2: Obtain graph G' from G by first including in G all in-edges of G (designated as in-edges of G') and not including any out-edges. Each of the remaining, neutral edges of G is included in G' independently with probability $p = 1/(\log^{(i-1)} m)^b$.

Step 3: Call FINDFOREST($G', i-1$).

Step 4: Call FILTER(G, G'), designating some edges of G as out-edges.

Step 5: Call FINDFOREST($G, i-1$).

The procedure FINDFOREST depends on two subprocedures, BASECASE and FILTER. The *correctness condition for* BASECASE(G) is:

> If G satisfies the in-edge and out-edge soundness conditions then after the call BASECASE(G), G satisfies the completeness condition and the soundness conditions.

The *correctness condition for* FILTER(G, G') is as follows:

> Suppose G' is a subgraph of G, and G and G' satisfy the in-edge and out-edge soundness conditions. Suppose moreover that G' satisfies the completeness condition. Then the edges of G designated as out-edges by FILTER satisfy the out-edge soundness condition.

We give the details of BASECASE and FILTER later, and show they satisfy their correctness conditions. Assume for now that these conditions hold.

3.1 Correctness of FINDFOREST

Finally, the correctness invariant for FINDFOREST(G, i) is as follows:

> Suppose G satisfies the in-edge and out-edge soundness conditions. Then after the call FINDFOREST(G, i), G satisfies the completeness condition and the soundness conditions.

We show by induction that the algorithm satisfies the above invariant.

Suppose that G satisfies the soundness conditions, and consider the call FINDFOREST(G, i). If $i = 2$ then it follows from the correctness condition for BASECASE that after the call BASECASE(G), G satisfies completeness and soundness.

Now suppose $i > 2$. First consider the Borůvka steps executed in Step 1 of FINDFOREST. These steps designate some MSF edges as in-edges, preserving the soundness conditions.

Next consider the graph G' obtained from G in Step 2. Since G' is a subgraph of G', every MSF edge of G that appears in G' is also an MSF edge of G'. The in-edges of G' are precisely the in-edges of G, which are MSF edges of G by the in-edge soundness of G. Thus the in-edges of G' are MSF edges of G', so in-edge soundness holds for G'. Out-edge soundness trivially holds for G' since G' has no out-edges.

Next consider G' after the call FINDFOREST$(G', i-1)$. By the inductive hypothesis, G' satisfies completeness and soundness.

Next in Step 4 there is a call FILTER(G, G'), designating some of the edges of G as out-edges. By the correctness condition of FILTER, these edges satisfy the out-edge soundness condition.

We have seen that at the beginning of Step 5, G satisfies the soundness conditions. By the induction hypothesis, therefore, after the call FINDFOREST$(G, i-1)$ in that step, G satisfies completeness and soundness.

3.2 Bounds on graph parameters

Now we consider the resource requirements of FINDFOREST. The first step in the analysis is to prove bounds on the number of neutral edges and the number of in-edge trees in graphs arising at different levels of recursion. Consider a call FINDFOREST(G, i). We claim that the number of active in-edge trees in G is at most $m/(\log^{(i)} m)^a$ and the expected number of neutral edges in G is at most $m/(\log^{(i)} m)^b$. The claim is trivially true for the initial call, since for that call $i = \log^* m$, so $log^{(i)} = 1$. We show that an invocation FINDFOREST(G, i) preserves the truth of the claim in its recursive invocations. Each Borůvka step in Step 1 reduces by a constant factor the expected number of active in-edge trees in G. Since Step 1 performs $O(\log[(\log^{(i-1)} m)^a])$ such steps, by choice of the constant hidden by the big Oh, the expected number of trees after the step is at most the number before the step divided by $2(\log^{(i-1)} m)^a$, which in turn is certainly at most $m/(\log^{(i-1)} m)^a$. Now we consider the number of edges.

The neutral edges of G' are obtained from the neutral edges of G by sampling with probability $p = 1/(\log^{(i-1)} m)^b$. Thus the expected number of neutral edges in G' is p times the number of neutral edges in G. The number of edges in G is certainly at most m, so the expected number of neutral edges in G' is at most pm, which is $m/(\log^{(i-1)} m)^b$. Thus the recursive call in Step 3 satisfies the claim.

Next we consider the recursive call in Step 5. We use a bound on the number of edges not designated out-edges in Step 4. The following lemma is a generalization of the lemma at the heart of the analysis of a linear-time randomized sequential algorithm for minimum spanning trees [19].

Lemma 2 *Let n be the number of in-edge trees in G. After the call* FILTER(G, G')*, the expected number of neutral edges in G is at most $2n/p$.*

Since $n \le m/2(\log^{(i-1)} m)^a$ and $p = 1/(\log^{(i-1)} m)^b$, we infer that the expected number of neutral edges in G after Step 4 is at most $m/(\log^{(i-1)} m)^{a-b}$, which in turn is at most $m/(\log^{(i-1)} m)^b$ since $a \ge 2b$.

3.3 Analysis of resource requirements of FINDFOREST

The depth of the recursion is $\log^* m$. Hence the number of invocations is $2^{\log^* m}$. In each invocation the dominant step is the call to FILTER. We show later that the time required by FILTER is $O(\log m \log\log\log m / \log\log m)$, and the work is linear. Thus the total time is $O(2^{\log^* m} \log m \log\log\log m / \log\log m)$, which is $o(\log m)$.

Now we bound the work done. For each invocation, the work done is linear in the number of neutral edges. Hence at each level of recursion (each value of i), the work done is linear in the number of neutral edges in graphs G that appear as arguments to invocations at that level. There is one top-level call, two calls at the next level, four at the next, and so on. Using the expected bounds on the sizes of graphs that were derived in the previous subsection, we infer that the total expected work is $\sum_i 2^i O(m/(\log^{(i)} m)^b$, which is linear.

4 The subprocedure BASECASE

We describe the subprocedure BASECASE(G) used in Step 0 of FINDFOREST for the base case of the recursion. In this case, it is assumed of the graph G that the number of trees in the forest of in-edges is small compared to the number of vertices in the original graph, and that the number of neutral edges is small compared to the number of edges in the original graph. Hence we can afford to use a fairly inefficient algorithm for this case.

The subprocedure also constructs an auxiliary graph, used by the subprocedure FILTER, consisting of a path P_T for each tree T of edges that are in-edges at the beginning of the call. Each such path is called the *trunk* of T.

Let $\hat{G}$ be the graph obtained from G by contracting all the in-edges. As mentioned in Subsection 2.1, the algorithm keeps track of the components of in-edges and the edges between them, so the contraction step is trivial. Each tree T of in-edges is contracted to a vertex v in $\hat{G}$, called the *target* of T. For each such vertex, we calculate an k-min tree T_v of v. (If v's i^{th} min edge is an out-edge, we let T_v be the $(i-1)$-min tree of v.) Finally, for each edge in such a tree T_v, we designate the corresponding edge in G as an in-edge.

To find the trees T_v, we proceed as follows. Replace each edge xy of $\hat{G}$ with two oppositely directed arcs, $x \to y$ and $y \to x$. Next, for each vertex x determine the k cheapest outgoing arcs $x \to y$. Let $\bar{G}$ be the graph consisting of the union over all vertices x of the k cheapest outgoing arcs of x. Next, for each vertex v we execute the following variant of Prim's algorithm. Initialize the set S_v to $\{v\}$. Initialize the trunk to consist only of the vertex v. Repeat the following step k times: find the cheapest arc $x \to y$ outgoing from S_v. Append it to the end of the trunk. If it is an out-edge, halt. Otherwise, designate the corresponding edge xy as belonging to T_v, insert y into S_v, and repeat.

Since S_v never contains more than $k+1$ vertices and each vertex has at most k outgoing arcs, the minimum can be found in $O(k^2)$ time. Thus the above loop takes $O(k^3)$ time using one processor per vertex of $\hat{G}$.

Lemma 3 BASECASE(G) *satisfies its correctness condition.*

Proof sketch: We assume that G satisfies in-edge and out-edge soundness before the call to BASECASE. By in-edge soundness, the in-edges of G belong to the MSF. It follows that the MSF of $\hat{G}$ is contained in the MSF of G. Each edge designated as an in-edge by the procedure belongs to the MSF of $\hat{G}$ and hence of G. Thus in-edge soundness is preserved. By out-edge soundness of G before the call, for every vertex v, the k-min tree of v is contained among the in-edges and neutral edges of G before the call. Let w be the vertex into which v is coalesced by the contractions It is straightforward to show that each edge of v's k-min tree that is neutral before the call belongs to T_w and is therefore designated an in-edge. Thus the call achieves soundness. □

5 The FILTER subprocedure

The goal of FILTER(G, G') is to identify edges in G that are not needed for completeness and can therefore be designated as out-edges without violating out-edge soundness. To facilitate this task, the subprocedure uses the forest of in-edges selected in the sample graph G' during the recursive call FINDFOREST($G', i+1$) in Step 3. More precisely, FILTER uses the trunks constructed during the call to BASECASE(G') and a forest, called the *merge forest*, described in the next subsection, that is constructed during the Borůvka steps. We refer to the *nodes* of the trunks and the merge forest to distinguish them from the vertices of the graphs.

5.1 The merge forest

We define a rooted forest M, called the *merge forest*, that captures the effect of the Borůvka steps performed during a call to FINDFOREST. This structure resembles closely the Borůvka tree defined and used by King [21] for verification of minimum spanning trees. (The differences reflect our modification of the Borůvka step.)

The nodes of M correspond to in-edge trees in the graph, and we use ϕ to denote the mapping from in-edge trees to nodes of M. For each in-edge tree T arising during a call to FINDFOREST, there is a node $\phi(T)$. If during a Borůvka step some in-edge trees $T_1, T_2, \ldots, T_t$ are merged to form a tree T because some edges between them are designated in-edges, then $\phi(T_1), \ldots, \phi(T_t)$ are the children of $\phi(T)$ in the merge forest M. For $i = 1, \ldots, t$, if the new in-edge e incident to T_i is the cheapest edge incident to T_i then the edge in M from $\phi(T_i)$ to its parent $\phi(T)$ is assigned the cost of e; otherwise the edge in M is assigned cost negative infinity. Construction of the merge forest can be incorporated into the implementation of Borůvka steps.

Note that the depth of the merge forest for a graph is bounded by the number of Borůvka steps executed on that graph. The number of Borůvka steps at level i is $O(\log[(\log^{(i-1)} m)^a])$, which is $O(\log^{(i)} m)$, so the total number of Borůvka steps executed on a graph is

$$O(\log^{(3)} m + \log^{(4)} m + \cdots)$$

which is $O(\log^{(3)} m)$.

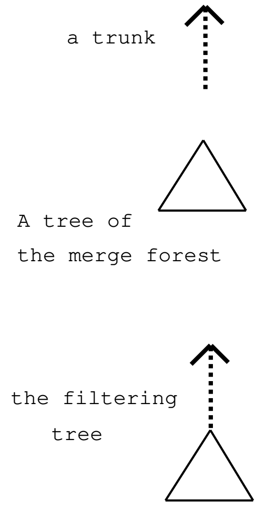

Figure 1: A filtering tree is obtained by attaching a stem to a tree of the merge forest.

5.2 The Filtering Forest

The subprocedure FILTER(G, G') builds a structure from the merge forest M and trunks resulting from the recursive call FINDFOREST($G', i+1$) in Step 3. For each rooted tree T in the merge forest, the filtering structure contains the rooted tree obtained from T by attaching the first node of a trunk P to the root of T. The trunk P used is the trunk associated with the in-edge tree corresponding to the root of T. We call the resulting tree a *filtering tree*, and the collection of these trees is called the *filtering forest*.

We define a kind of least-common ancestor for the filtering forest. Let F_0 be the forest of in-edges before the call FINDFOREST($G', i+1$) in Step 3, and let F be the forest after the call. For a vertex v of G', let $F_0(v)$ denote the tree of F_0 that contains v, and similarly define $F(v)$. Let $P(v)$ denote the trunk whose first node is the target[2] of $F(v)$.

For vertices v and w of G', we define $\mathrm{anc}_v(w)$ according to the following three cases (depicted in Figure 5.2). If $F(w) = F(v)$ then $\phi(F_0(v))$ and $\phi(F_0(w))$ are nodes of the same rooted tree of the merge forest M. In this case, $\mathrm{anc}_v(w)$ is defined to be the least common ancestor of these two nodes. Assume $F(w) \neq F(v)$. Suppose that the target of $F(w)$ appears as a node in the trunk $P(v)$. In this case, $\mathrm{anc}_v(w)$ is defined to be that node. Finally, suppose that the target of $F(w)$ does not appear in the trunk $P(v)$. In this case, $\mathrm{anc}_v(w)$ is the last node of the trunk attached to $M(v)$, i.e. the root of the corresponding filtering tree. Note that in each of the tree cases, $\mathrm{anc}_v(w)$ is a ancestor of $\phi(F_0(v))$ in the filtering forest.

Define $c_v(w)$ to be the maximum cost of an edge on the path in the filtering forest from $\phi(F_0(v))$ to $\mathrm{anc}_v(w)$.

Lemma 4 *Suppose that after the call* FINDFOREST($G', i+1$) *in Step 3, G' satisfies soundness and completeness. Let vw be an edge of G. If the cost of vw exceeds* $\max c_v(w), c_w(v)$ *then vw is not in the k-min tree of any node in G.*

The procedure FILTER designates as out-edges all those edges of G that satisfy the condition of Lemma 4.

[2] That is, the result of contraction in BASECASE. See Section 4.

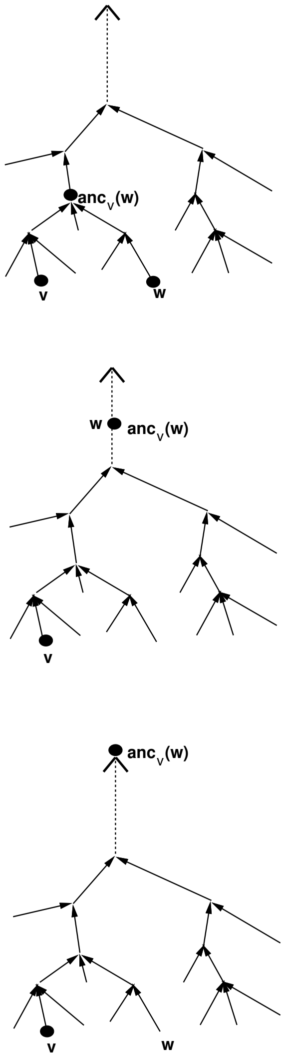

Figure 2: The three cases in the definition of $\text{anc}_v(w)$. In the first case, v and w map to nodes of the same merge tree. In the second case, w maps to a node on the trunk of the merge tree containing v. In the third case, w does not appear on either v's merge tree or on the attached trunk. combined tree is obtained by attaching a stem to a tree of the merge forest.

5.3 Efficient implementation of Filter

We describe how to efficiently identify all edges vw that by Lemma 4 can be designated as out-edges. First process the filtering forest so that $\text{anc}_v(w)$ can be determined in constant time for any vertices v and w.

5.3.1 Structures for calculating $\text{anc}_v(w)$

This step consists in processing the trunks and processing the merge forest. For each trunk, build a perfect hash table of the nodes comprising it. This can be done in time proportional to the size of the trunk. (This step can be done once during BaseCase(G').)

For the merge forest, build a least-common ancestor structure [14, 27] for M so that, given a pair of nodes x and y, the least common ancestor of x and y in M can be determined in constant time. We use the structure proposed by Schieber and Vishkin, but to construct it we use an algorithm whose running time depends on the height D of M. It is straightforward to adapt their algorithm to run in time $O(D \log n / \log\log n)$ and work $O(n)$, where n is the size of M. The main difficulty in the Schieber/Vishkin algorithm is to number the vertices in left to right order; but this can be done by means of a sweep up the trees to compute the size of the subtree of each node, followed by a sweep down to calculate the numbering. Each step in the up-sweep is a parallel prefix-sums computation.

Once these structures have been built, one can find $\text{anc}_v(w)$ in constant time. If $F(v) = F(w)$ then use the least-common-ancestor structure for the merge forest. If $F(v) \neq F(w)$ then use the hash table associated with the trunk $P(v)$ to determine if the target of $F(w)$ occurs in $P(v)$. If so, that occurrence is $\text{anc}_v(w)$. If not, then $\text{anc}_v(w)$ is the last node of $P(v)$.

5.3.2 Processing the lengths on trunk edges

Next, build a table for each of the trunks by traversing its edges, first to last. For the i^{th} edge, record the maximum cost among edges 1 through i. This step takes time proportional to the size of the trunk. (This step can also be done during BaseCase(G').)

5.3.3 Determining costs on leaf-to-root paths of the merge forest

Next, build a table for each node of the merge forest by scanning down the merge forest starting at the roots. For each node, record the maximum cost on the path from the root to that node.

5.3.4 Determining $c_v(w)$: the easy case

For each edge vw, if $\text{anc}_v(w)$ belongs to $P(v)$ then $c_v(w)$, the maximum cost on the path from $\phi(F_0(v))$ to $\text{anc}_v(w)$, can be determined in constant time by consulting the table for the merge forest and the table for the trunk $P(v)$.

5.3.5 Determining $c_v(w)$: the hard case

Use a parallel version of King's algorithm [21] to determine $c_v(w)$ for each edge vw in G such that $\phi(F_0(v))$ and $\phi(F_0(w))$ occur in the same tree of the merge forest. King's algorithm consists primarily of scanning down the forest, assigning labels to the nodes. The time per node is $O(\log\log n)$. At each level, the work needs to be rebalanced among the processors but this can be done in $O(\log\log n)$ time using standard techniques. The total time is thus $O(D\log\log n)$, where D is the depth of the merge forest. The total work is linear in the number of edges that need to be checked, which is bounded by the number of neutral edges of G.

5.3.6 Designating out-edges

Finally, designate edges vw of G as out-edges in accordance with Lemma 4: if the cost of vw exceeds $c_v(w)$ and $c_w(v)$ then vw is designated an out-edge. To maintain the representation of out-edges, calculate for each in-edge tree the cost of the cheapest incident out-edge. This can be done in $O(\log\log n)$ time and linear work using the minimum-finding algorithm of Shiloach and Vishkin [28].

5.3.7 Resource requirements

The most time-consuming step is constructing the least-common ancestor structure. This takes time $O(D\log n/\log\log n)$ and $O(n)$ work, where n is the size of the merge forest M, and D is its depth. We showed that $D = O(\log^{(3)} m)$, and of course n is the number of in-edge trees in G' before the recursive call FindForest$(G', i+1)$ in Step 3. The total time for Filter(G, G') is therefore $O(\log n\log\log\log m/\log\log n)$. The work is $O(n+m)$, where m is the number of neutral edges.

6 The Phase II Algorithm

The Phase II algorithm, FinishUp(G), is much simpler. It invokes a recursive procedure that resembles FindForest; however, each recursive call finds a minimum spanning forest. Edges can therefore be simply deleted instead of being designated out-edges. Furthermore, the recursion depth is constant. Here is the procedure FinishUp(G):

Step 0: Let G' be obtained from G by including each edge of G independently with probability $p = 1/\sqrt{k}$, where k is as specified in the top-level algorithm.

Step 1: Call Basic$(G', 3)$ to obtain the MSF of G'.

Step 2: Use the minimum spanning forest of G' to determine some edges of G that do not belong to the MSF of G, and delete these edges from G.

Step 3: Call Basic$(G, 3)$ to find the MSF of G.

The choice of edges to delete in Step 2 is based on a simpler condition than that used in Filter. For an edge vw of G, if there is a path in the MSF of G' that connects v to w, and every edge on this path is cheaper than vw, then vw does not belong to the MSF of G. Dixon, Rauch, and Tarjan [7] have given a parallel algorithm to implement this check for all edges of G in logarithmic time and linear work.

As in the main algorithm, we let m denote the number of edges in the original input graph G_0. Then at the beginning of FinishUp(G), the graph G certainly has at most m edges, so the expected number of edges in the graph G' is mp, which is $m/\sqrt{k}$. It follows from the main lemma of [19] that the expected number of edges in G after the deletions in Step 3 is at most n/p, where n is the number of vertices in G. As we showed in Section 3, $n \le m/k$, so the number of edges in G after the deletions is at most $m/\sqrt{k}$.

Now we give the recursive procedure Basic(G, i). We assume that on entry the expected number of neutral edges in G is $O(m/\log^{(i)} m)$. This holds for the call Basic$(G, 3)$ in Step 3 of FinishUp. We show in the procedure that consequently this invariant holds for recursive invocations as well.

Step 0: Perform $\Theta(\log^{(i)} m)$ Borůvka steps. The resulting graph has expected $O(m/(\log^{(i-1)} m)^2)$ in-edge trees. If $i = 1$ then the in-edges selected comprise the MSF of G; return in this case.

Step 1: Obtain G' from G by randomly including each edge independently with probability $p = 1/\log^{(i-1)} m$. The sample graph has expected $O(m/\log^{(i-1)} m)$ edges.

Step 2: Recursively call Basic$(G', i-1)$ to designate as in-edges all the remaining MSF edges of G'.

Step 3: As in Step 2 of FinishUp, use the MSF of G' to determine which edges of G to delete. This takes $O(\log m)$ time and expected $O(m/\log^{[i]} m)$ work. By the main lemma of [19], the expected number of remaining edges is the number of in-edge trees times $1/p$. This product is $O(m/\log^{(i-1)} m)$.

Step 4: Recursively call Basic$(G, i-1)$ to designate as in-edges all the remaining MSF edges of G. of G.

The work done by this procedure is linear in each invocation. The time required is logarithmic. The number of recursive invocations resulting from the top-level call Basic$(G, 3)$ in Step 3 of FinishUp is seven. Thus the total time is logarithmic and the work is linear.

References

[1] B. Awerbuch and Y. Shiloach, "New connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM," *IEEE Transactions on Computers,* C-36 (10), 1987, pp. 1258-1263.

[2] O. Borůvka, "O jistém problému minimálním, *Práca Moravské Přírodovědecké Společnosti* 3, 1926, pp. 37-58. (In Czech.)

[3] F. Y. Chin, J. Lam, and I.-N. Chen, "Efficient parallel algorithms for some graph problems, *CACM 25*, 1982, pp. 659–665.

[4] K. W. Chong and T. W. Lam, "Finding connected components in $O(\log n\log\log n)$ time on the EREW PRAM, *Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms*, 1993, pp. 11–20.

[5] R. Cole, P. N. Klein, and R. E. Tarjan, "A linear-work parallel algorithm for finding minimum spanning trees," 6th Annual ACM Symposium on Parallel Algorithms and Architectures (1994), pp. 11-15.

[6] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree, and graph problems," *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science*, 1986, pp. 478-491.

[7] B. Dixon, M. Rauch, and R. E. Tarjan, "Verification and sensitivity analysis of minimum spanning trees in linear time," *SIAM J. on Computing* 21, 1992, pp. 1184-1192.

[8] B. Dixon and R. E. Tarjan, "Optimal parallel verification of minimum spanning trees in logarithmic time," *Algorithmica*, to appear.

[9] M. Fredman and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Proc. 31st Annual IEEE Symp. on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 478-491.

[10] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," SIAM Journal on Computing, 6(1991), 1046–1067.

[11] J. Gil, Y. Matias, U. Vishkin, "Towards a theory of nearly constant time parallel algorithms, *Proc. 36st Annual IEEE Symp. on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA,1991, pp. 698-710.

[12] S. Halperin, U. Zwick, "an optimal randomized logarithmic time connectivity algorithm for the EREW PRAM.", 6th Annual ACM Symposium on Parallel Algorithms and Architectures (1994) pp. 1-10.

[13] S. Halperin, U. Zwick, "Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems," *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1996, pp. 438-447.

[14] D. Harel and R.E. Tarjan, "Fast algorithms for finding nearest common ancestors," SIAM Journal on Computing, 1984, 338–355.

[15] "O jistém problému minimálním," *Práca Moravské Prírodovedecké Spolecnosti 6* (1930), pp. 57-63. (In Czech.)

[16] D. B. Johnson and Metaxas, "A parallel algorithm for computing minimum spanning trees, *Proc. 4th Annual ACM Symp. on Parallel Algorithms and Architectures*, 1992, pp. 3630–372.

[17] D. R. Karger, "Approximating, verifying, and constructing minimum spanning forests," manuscript, 1992.

[18] D. R. Karger "Random sampling in matroids, with applications to graph connectivity and minimum spanning trees," *Proc. 34st Annual IEEE Symp. on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 84-93.

[19] D. R. Karger, P. N. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," Journal of the ACM, Vol. 42 (1995), pp. 321-328.

[20] R. M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," Chapter 17 in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity* J. van Leeuwen, ed., MIT Press, Cambridge, Mass., 1990, pp. 869–941.

[21] V. King, " A simpler minimum spanning tree verification algorithm," *Proc. 4th International Workshop on Algorithms and Data Structures*, published as *Lecture Notes in Computer Science 955*, Springer-Verlag, Berlin, 1995, pp. 440-448

[22] V. King, personal communication, 1994.

[23] P. N. Klein and R. E. Tarjan, "A randomized linear-time algorithm for finding minimum spanning trees, to appear in *Proc. 26th Annual ACM Symp. on Theory of Computing*, 1994, pp. 9-15.

[24] N. Megiddo, "Parallel algorithms for finding the maximum and the median almost surely in constant time," Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, October 1982.

[25] P. Raghavan, "Lecture Notes on Randomized Algorithms," Research Report RC 15340 (#68237), Computer Science/Mathematics IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, 1990, p. 54.

[26] C. Savage and J. Ja'Ja, "Fast efficient parallel algorithms for some graph problems, *SIAM J. Comput 10* 1981, pp. 682.

[27] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplification and parallelization, *SIAM J. Comput. 17* (1988), PP.11253-1262.

[28] Y. Shiloach, U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," Journal of Algorithms, 1(1981), 88–102.

[29] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *J. Algorithms 3*, 1982, pp. 57-67.

[30] R. E. Tarjan, *Data Structures and Network Algorithms*, Chapter 6, Society for Industrial and Applied Mathematics, Philadelphia, 1983.