December 1995

UILU-ENG-95-2241
ACT-136

Applied Computation Theory

Algorithms for All Single Deletions in a Minimum Spanning Tree, Simulataneously

Bevan Das and Michael C. Loui

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
Unclassified	None

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
UILU-ENG-95-2241 ACT-136

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Coordinated Science Lab
University of Illinois

6b. OFFICE SYMBOL (If applicable)
N/A

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS (City, State, and ZIP Code)
1308 W. Main St.
Urbana, IL 61801

7b. ADDRESS (City, State, and ZIP Code)

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
National Science Foundation

8b. OFFICE SYMBOL (If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Algorithms for All Single Deletions in a Minimum Apanning Tree, Simulataneously

12. PERSONAL AUTHOR(S)
Bevan DAs and Michael C. Loui

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	December 1995	31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
minimum spanning tree, edge deletion, node deletion, graph algorithm, parallel algorithm

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Let T be a minimum spanning tree of a biconnected graph $G = (V, E)$ with weighted edges. Let $m = |E|$ and $n = |V|$. We present sequential and parallel algorithms for determining the minimum spanning tree of each graph $G - t$ for all edges t in T and each graph $G - v$ for all v in V, simultaneously.

For an edge t in T, define the *replacement* for t to be the minimum weight edge e that connects the two components of T after t is deleted. For a node v in T, define the *replacement* for v to be the minimum weight set of edges, $R(v)$, that connect the $\delta_T(v) - 1$ components of T after v is deleted, where $\delta_T(v)$ is the tree degree of v. Our sequential algorithms for finding all edge replacements and for finding all node replacements run in $O(m)$ and $O(m + n\alpha(n))$ time, respectively, assuming that the edges are sorted by weight. Our parallel algorithms for these problems run in $O(\log n)$ and $O(\log^2 n)$ time, respectively, using m processors on a CREW PRAM.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

UNCLASSIFIED

Algorithms for All Single Deletions in a Minimum Spanning Tree, Simultaneously

Bevan Das and Michael C. Loui[1]

Department of Electrical and Computer Engineering
and Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
b-das@uiuc.edu, m-loui@uiuc.edu

December 6, 1995

Abstract

Let T be a minimum spanning tree of a biconnected graph $G = (V, E)$ with weighted edges. Let $m = |E|$ and $n = |V|$. We present sequential and parallel algorithms for determining the minimum spanning tree of each graph $G - t$ for all edges t in T and each graph $G - v$ for all v in V, simultaneously.

For an edge t in T, define the *replacement* for t to be the minimum weight edge e that connects the two components of T after t is deleted. For a node v in T, define the *replacement* for v to be the minimum weight set of edges, $R(v)$, that connect the $\delta_T(v) - 1$ components of T after v is deleted, where $\delta_T(v)$ is the tree degree of v. Our sequential algorithms for finding all edge replacements and for finding all node replacements run in $O(m)$ and $O(m + n\alpha(n))$ time, respectively, assuming that the edges are sorted by weight. Our parallel algorithms for these problems run in $O(\log n)$ and $O(\log^2 n)$ time, respectively, using m processors on a CREW PRAM.

Key words: minimum spanning tree, edge deletion, node deletion, graph algorithm, parallel algorithm.

[1]Both supported by the National Science Foundation under Grant CCR-9315696.

1 Introduction

We consider two problems in this paper: All Edge Replacements (AER) and All Node Replacements (ANR). Informally, the problem AER considers all single edge deletions in a minimum spanning tree (MST); the problem ANR considers all single node deletions in a MST. Let undirected graph $G = (V, E)$ have n nodes and m edges, and let T be the MST of G, where $w(e)$ is the *weight* of edge e. For a single edge deletion, let $G - e$ denote the graph $(V, E - e)$. Similarly, let $G - v$ denote the graph after node v is deleted, that is, $(V - v, E - \{(u, v) | u \in V\})$. Formally, the problem AER is to determine the MST for each graph $G - e$, for all e in E simultaneously (a total of m MSTs). Likewise, the problem ANR is to determine the MST for each graph $G - v$, for all v in V simultaneously (a total of n MSTs).

We assume that G is biconnected, or equivalently, that G is connected and has no articulation points. As a result, each graph $G - e$ or $G - v$ is connected. In the closing comments in Section 6, we discuss how the algorithms are affected if G is not biconnected.

We also assume, without loss of generality, that the edge weights are distinct. If two different edges have the same weight, then we break ties lexicographically: for an edge $e = (u, v)$, we use the triple $(w(e), \min(u, v), \max(u, v))$ in the tie breaker. Consequently, the MST of each graph is unique.

The main results of this paper are four algorithms, two sequential and two parallel. The two sequential algorithms, Find Edge Replacements (FER) and Find Node Replacements (FNR), solve AER and ANR, respectively. Likewise, the two parallel algorithms, Find Edge Replacements–Parallel (FERP) and Find Node Replacements–Parallel (FNRP), solve AER and ANR, respectively.

FER finds all edge replacements in $O(m)$ time sequentially, assuming that the edges have been sorted by weight by the algorithm that determined T. (For example, the Prim-Dijkstra MST algorithm [Pr 57] sorts the edges as it grows the MST.) The *replacement* for a tree edge t is $r^*(t)$, the minimum weight nontree edge that connects the two components of $T - t$. A nontree edge e has no replacement, because T is the MST of $G - e$. FER determines $r^*(t)$ for each tree edge t in the following manner: add the nontree edges to T in order of increasing weight $w(e_1) < w(e_2) < \ldots$; form and contract the fundamental cycle $C(e_k)$ for each e_k; and stop when there are no more edges to consider.

FNR, like FER, forms and contracts the cycles $C(e_k)$ for the nontree edges in order of increasing weight. With the edges sorted previously by weight, FNR finds all node replacements in $O(m + n\alpha(n))$ time, where $\alpha(n)$ is the inverse of Ackermann's function; if the edges are not sorted by weight, FNR takes $O(m \log n)$ time. The *replacement* for a node v is $R^*(v)$, the minimum weight set of nontree edges that connects the components of $T - v$. Since there are $\delta_T(v)$ components of $T - v$, where $\delta_T(v)$ is the degree of v in T, $R^*(v)$ has $\delta_T(v) - 1$ edges. Consequently, for a leaf y, $R^*(y)$ has zero edges, that is, $R^*(y) = \emptyset$.

The node replacement $R^*(v)$ is the MST of a multigraph $CG(v)$ [CH 78], which we call the *component graph* of v. Each component of $T - v$ induces a subgraph in $G - v$; $CG(v)$ is formed by replacing each component with one node. Intuitively, FNR simulates Kruskal's algorithm [Kr 56] on each $CG(v)$, since Kruskal's version of the greedy MST algorithm considers nontree edges in order of increasing weight. To ensure that the simulation is correct, FNR initially replaces each node v with a star of $\delta_T(v)$ nodes. FNR then proceeds

in the manner of FER.

The parallel algorithm FERP finds all edge replacements in $O(\log n)$ time, using m processors on a CREW PRAM. FERP is similar to a parallel ear decomposition algorithm of Maon, Schieber, and Vishkin [MSV 86]: Both FERP and the algorithm of Maon *et al.* use parallel tree contraction to partition the tree edges.

FNRP finds all node replacements in parallel in $O(\log^2 n)$ time, using m processors on a CREW PRAM. FNRP combines the ideas of FNR with the mechanism of FERP. Like FNR, FNRP simulates a greedy MST algorithm on each $CG(v)$: FNRP uses FERP to find the minimum outgoing edge from each connected component in $CG(v)$. Specifically, FNRP simulates Sollin's algorithm [BGH 65], which constructs the MST of a graph in $O(\log n)$ iterations. Because each iteration runs FERP, each iteration takes $O(\log n)$ time, hence the total time for FNRP, $O(\log n \times \log n) = O(\log^2 n)$ time.

The most significant results in this paper are FNR and FNRP. FNR provides the best time bound for finding all node replacements sequentially, except when the graph is dense. For most graphs, where $m = o(n^2)$ with edges sorted or $m = o(n^2/\log n)$ without edges sorted, FNR improves upon the $O(n^2)$ time of the algorithm of Chin and Houck [CH 78], which also found all node replacements. FNRP fills a gap: whereas sequential algorithms for ANR and parallel algorithms for AER have been presented elsewhere, no parallel algorithm for ANR has previously been published.

Furthermore, FERP provides two techniques that may be useful in the design of other parallel algorithms. First, we show how to handle vector-valued inputs in the parallel tree contraction; usually, each node has only one input value. Second, we represent an $O(n^2)$ size lookup table implicitly with just $O(m)$ values. This smaller size lookup table reduces the number of processors required from n^2 to m.

In addition, all the algorithms in this paper increase the reliability of other algorithms that use MSTs during execution. For example, information broadcast, network synchroniztion, and deadlock resolution may use MSTs as a basic building block. For these algorithms, it can be useful to reconnect a MST after an edge or node deletion.

FER and FERP are not the best known algorithms to find all edge replacements. We present FER because of its simplicity and its relation to FNR. We include FERP since it is used in FNRP and it introduces two general techniques for the design of parallel algorithms. For the best sequential time bound, Dixon, Rauch, and Tarjan present a sequential algorithm that verifies MSTs in $O(m)$ time [DRT 92]; combined with a graph transformation of Tarjan [Ta 79, p. 713], the algorithm of Dixon *et al.* finds all edge replacements in $O(m)$ time. Their algorithm does not require that the edges are sorted by weight while T is determined. (For example, the Fredman-Tarjan MST algorithm [FT 87] does *not* sort the edges.) In comparison, FER runs in $O(m \log n)$ time if the edges are not sorted by weight.

Similarly, for the lowest parallel work bound, Dixon and Tarjan describe a parallel algorithm to verify MSTs in $O(\log n)$ time using $\Theta((m+n)/\log n)$ processors on a CREW PRAM [DT 94]. This parallel algorithm of Dixon and Tarjan, combined with the graph transformation of Tarjan [Ta 79], finds all edge replacements in $O(\log n)$ time with $O(m+n)$ work, which is optimal.

The rest of this paper discusses previous research related to MST updates and presents the technical details of the algorithms. Section 3 describes the sequential algorithms FER and FNR, Section 4 describes the parallel algorithm FERP, and Section 5 describes the

parallel algorithm FNRP.

2 Related Research

Chin and Houck provide the basic framework for AER and ANR [CH 78]. They present $O(n^2)$ time sequential algorithms for both AER and ANR. Furthermore, they prove that $T - t + r^*(t)$ is the MST of $G - t$ and that $T - v + R^*(v)$ is the MST of $G - v$, where the $+$ indicates a union with the edges of T.

As noted in Section 1, the time bound for finding edge replacements has been improved. Dixon *et al.* present an $O(m)$ time sequential algorithm [DRT 92], and Dixon and Tarjan present an optimal work $O(\log n)$ time parallel algorithm on the CREW PRAM [DT 94]. Both algorithms use the graph transformation of Tarjan [Ta 79].

Single edge update algorithms lead to naive algorithms that compare favorably for dense graphs. A naive algorithm for AER is to perform a single edge update $n-1$ times, one for each tree edge. For sequential algorithms, after $O(m)$ time for preprocessing the graph, the single edge update algorithm of Frederickson [Fr 85], combined with the sparsification technique of Eppstein *et al.* [EGIN 92], takes $O(\sqrt{n}\log(m/n))$ time per update. Thus, this naive algorithm takes $O(n\sqrt{n}\log(m/n))$ time to solve AER. Our FER algorithm is faster than the naive algorithm on sparse and moderately dense graphs, with $m = o(n\sqrt{n}\log(m/n))$.

For parallel algorithms, Das and Ferragina [DF 94] present an $O(\log n)$ time algorithm that takes $O(n^{2/3})$ work on an EREW PRAM for a single edge update of an MST. Applied to each tree edge, a naive algorithm based on [DF 94] takes $O(\log n)$ time with $O(n^{5/3})$ work on an EREW PRAM. FERP requires less work on sparse and moderately dense graphs, with $m = o(n^{5/3})$.

As stated previously, FERP is based on the parallel ear decomposition algorithm of Maon *et al.* [MSV 86]. In related work, Pawagi and Kaser [PK 93] present parallel algorithms for single and multiple edge and node deletions, each using $n^2/\log n$ processors on a CREW PRAM. The time bounds for the single edge deletion, k edge deletion, single node deletion, and k node deletion algorithms are, respectively, $O(\log n)$, $O(\log n + \log^2 k)$, $O(\log n + \log^2 \delta_T(v))$, and $O(\log n + \log^2 \mu)$, where $\mu = \min\{1 + \sum_{v \in \text{ deleted set}} \delta_T(v), n\}$. However, the algorithms of Pawagi and Kaser cannot be used to efficiently solve AER and ANR. For example, the k edge deletion algorithm handles k edge deletions from a single MST, while FERP produces output representing $n-1$ different MSTs.

AER and ANR consider only single edge and single node deletions; several results have been published for more general MST updates. There are several distributed algorithms for computing and maintaining MSTs; Pasquini and Loui [PL 94] give an overview of these algorithms, in addition to presenting a fault-tolerant distributed MST algorithm. Rauch Henzinger and King [RHK 95] present several randomized connectivity algorithms that take polylogarithmic time per update, including algorithms for maintaining a MST for a k-weight graph, approximating a MST in a graph with weights in a specific range, and maintaining a maximal spanning forest decomposition of order k. Finally, Kapoor and Ramesh [KR 95] present an algorithm for enumerating spanning trees of a graph in order of increasing weight, in $O(N\log n + mn)$ time, where N is the number of spanning trees in the graph.

3 Sequential Algorithms

In the sequential algorithms, the MST for each graph $G - e$ and $G - v$ is represented by the replacement edge or replacement set of edges. The FER algorithm uses the variable $r(t)$ for t's edge replacement. Initially, $r(t)$ is set to `null`; after FER terminates, $r(t) = r^*(t)$, where $r^*(t)$ is the correct edge replacement for t, defined in Section 1. Similarly, FNR computes a set $R(v)$ (defined in Section 3.2) to be v's node replacement. Section 3.2.7 shows that $R(v) = R^*(v)$, where $R^*(v)$ is the correct node replacement for v, defined in Section 1.

3.1 FER–Find Edge Replacements

In essence, FER considers cycles corresponding to nontree edges e_k, one per stage k, in order of increasing edge weight. In stage k, the algorithm sets the $r(t)$ for each tree edge t in the current cycle to e_k. To indicate that these tree edges have had their replacements set, FER *contracts* the cycle; the nodes in the current spanning tree that are visited by the cycle are grouped into one *supernode* in the spanning tree for the next stage.

3.1.1 Overview of FER

As a precondition, FER assumes that the MST T for G has been found, and that the nontree edges $e_1, \ldots, e_{m-n+1}$ have been sorted by weight, $w(e_1) < w(e_2) < \cdots < w(e_{m-n+1})$. For the remainder of the paper, the subscript k of e_k indicates that e_k is the kth minimum weight nontree edge. If the MST algorithm does not also sort the edges, then sorting the edges requires only time $O(m \log m)$ precomputation.

At the beginning of stage k, the supernodes and the edges that have not been contracted form a multigraph G_k. Let T_k be the MST of G_k. Each edge in T_k corresponds to an edge in T. For example, suppose $t = (y, z)$ is in T and y and z are not contained in the same supernode in T_k. Let y be in supernode y' in T_k and z be in supernode z' in T_k; the edge between y' and z' in T_k corresponds to t (G_k might also have other, nontree edges between y' and z'). Similarly, each nontree edge in G_k corresponds to a nontree edge in G. For ease of notation, we refer to a nontree edge of G_k as e, instead of writing "the edge in G_k corresponding to the nontree edge e in G."

The notation $C_k \cap T_k$ in this context means the edges of T_k that are in C_k, in other words, $C_k - e_k$. Suppose that $e_k = (u, v)$, where u and v are nodes contained in supernodes in T_k. We write that a node y is in $C_k - \{u, v\}$ if y is contained in some supernode in C_k and y is neither u nor v.

The invariant and termination conditions for FER formalize the details of each stage. Let $r_k(t)$ be the value of $r(t)$ at the beginning of stage k. The invariant for FER is that immediately before stage $k, 1 \leq k \leq m - n + 1$, $r_{k-1}(t) = r^*(t)$ if and only if $r^*(t) = e_j$ for some $j < k$. In words, just before stage k, $r(t)$ is correct for every tree edge t that has some edge e_j as its replacement, where $j < k$. During stage k, $r(t)$ is set to e_k for each t in $C_k \cap T_k$. To mark that the edge replacements for these tree edges have been found, FER contracts $C_k \cap T_k$ into one supernode in T_{k+1}. Hence, FER terminates after stage k if $k = m - n + 1$ (no more nontree edges to consider) or if T_{k+1} has only one supernode (no more tree edges to consider). In general, FER may terminate before stage $m - n + 1$.

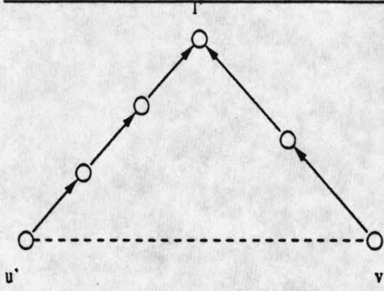

Figure 1: Constructing cycle from lowest common ancestor. Supernode $l' = \text{lca}(u', v')$ in T_k.

The pseudocode for FER is as follows:

Stage k
/* Derive T_{k+1} from T_k */
If endpoints of e_k are in same supernode in T_k, then
  $T_{k+1} := T_k$
Else
  Determine cycle C_k by finding l'
  For each $t \in C_k \cap T_k$, set $r(t) := e_k$
  Contract $C_k \cap T_k$, producing T_{k+1}

We will not formally argue the correctness of FER. It is simple to check that FER terminates. Furthermore, because all of $C_k \cap T_k$ is contracted during each stage k and because the nontree edges are considered in order of increasing weight, it is straightforward to verify that FER determines the correct replacement for each tree edge in T.

The next section explains how to determine the cycle C_k and how to contract $C_k \cap T_k$.

3.1.2 Determining and Contracting the Cycles

Let $e_k = (u, v)$ correspond to edge (u', v') in G_k. The cycle C_k consists of three portions: e_k, the path from u' to $\text{lca}(u', v')$, and the path from v' to $\text{lca}(u', v')$, where $\text{lca}(u', v')$ is the ancestor in T_k, not T. Let $l' = \text{lca}(u', v')$ in T_k, as shown in Figure 1.

In each stage, FER first checks whether u and v are contained in the same supernode, that is, whether $u' = v'$. If $u' = v'$, then $l' = u' = v'$ and $C_k = e_k$. Consequently, FER proceeds to the next stage and considers e_{k+1}. To check whether $u' = v'$, FER uses two *find* operations per edge, for a total of $O(m)$ *find* operations.

If $u' \neq v'$, FER determines l', then uses l' when contracting C_k. To determine l' given T_k, u', and v', FER traverses up T_k, alternating between a path from u' and a path from v': $p(u'), p(v'), p(p(u')), p(p(v')), \ldots$. FER stops at the first supernode that appears previously on the other path; this supernode is l'. For example, in Figure 1, $l' = p(p(v')) = p(p(p(u')))$, so the sequence stops at $p(p(p(u')))$. The supernode l' is found in at most $2|C_k \cap T_k|$ alternations, taking $O(|C_k \cap T_k|)$ *find* operations.

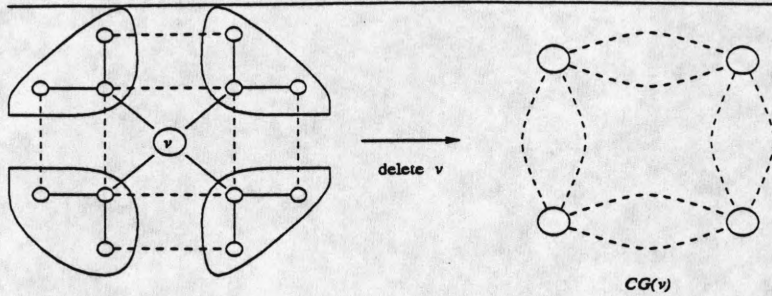

Figure 2: Component graph for node v, $\delta_T(v) = 4$.

Once l' is determined, FER contracts C_k in a similar fashion: from u' up to l' and from v' up to l'. The elementary operation in contracting C_k is contracting each tree edge. For each tree edge $t = (x', p(x'))$ in C_k, the contraction of t is the *union* of x' and $p(x')$. Hence, to contract t, FER uses one *find* operation (to determine $p(x')$) and one *union* operation. In total, FER contracts C_k using $O(|C_k \cap T_k|)$ *find* and *union* operations.

In summary, FER determines l' and contracts C_k for each stage in $\sum_k O(|C_k \cap T_k|)$ *find* and *union* operations. Because of the contractions, for $i \neq j$, $C_i \cap T_i$ has no edges in common with $C_j \cap T_j$. Hence,

$$\sum_k |C_k \cap T_k| = |T| = n - 1,$$

and FER uses $O(n)$ operations to determine and contract all the $C_k \cap T_k$. The disjoint set union algorithm of Gabow and Tarjan [GT 85] can perform these $O(n)$ operations in $O(n)$ time, if the *union* operations follow a predefined tree structure. Since FER only takes the *union* of a supernode and its parent in T_k, each *union* corresponds to an edge in T. Hence, FER uses T as the predefined tree structure. For example, FER requires that, given x' in T_k, $p(x')$ can be determined in a small amount of time; the data structure in the algorithm of Gabow and Tarjan provides this information in $O(1)$ time.

However, the total time for FER is $O(m)$, not $O(n)$. Because FER checks whether $u' = v'$ for each e_k, FER uses $O(m)$ operations total. The algorithm of Gabow and Tarjan performs these operations in $O(m)$ time. (On a side note, the lowest common ancestor algorithm of Harel and Tarjan [HT 84] can be used to determine l' for each stage in total time $O(n)$.)

3.2 FNR–Find Node Replacements

The Find Node Replacements (FNR) algorithm determines the node replacement for each v in V. The basic strategy of FNR is the same as FER's: consider the nontree edges in order of increasing weight, form a cycle in each stage, and contract some of the edges in that cycle. For FNR, we refer to the basic operation as *merging* the two endpoints of an edge instead of contracting the edge. Therefore, for FNR, we describe the contraction of each cycle in terms of which pairs of supernodes are merged.

Chin and Houck [CH 78] show that the edges in $R^*(v)$ correspond to the edges in the MST of a multigraph $CG(v)$, which we call the *component graph* for v. $CG(v)$ is formed from G by deleting v, and replacing each subgraph induced by a component of $T - v$ with a single node. Figure 2 shows $CG(v)$ for a node with $\delta_T(v) = 4$.

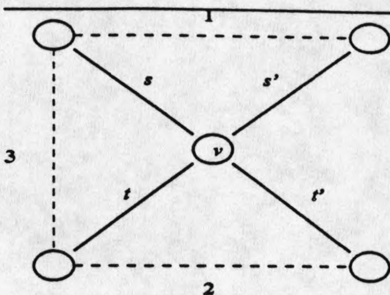

Figure 3: Counterexample to naive algorithm for ANR

To compute the MST of $CG(v)$ for each v separately takes too much time. Instead, FNR simulates Kruskal's algorithm [Kr 56], a greedy MST algorithm, on $CG(v)$ for all v simultaneously. In the process, FNR determines a set of edges $R(v)$; in Section 3.2.7, we show that $R(v) = R^*(v)$.

3.2.1 Flawed Naive Algorithms for ANR

Let $T(v)$ be the set of tree edges incident on v. A naive algorithm for ANR would set $R(v)$ equal to $\{r(t)|t \in T(v)\}$. However, for some edge t in $T(v)$, v might be an endpoint of $r(t)$; hence, $r(t)$ would not be in $R^*(v)$ (in fact, $r(t)$ would not be in $G - v$).

Let $E(v)$ be the set of edges in G incident on v. A less naive algorithm for ANR would modify the definition of $r(t)$ so that the replacement edge is not incident on v. In other words, the less naive algorithm defines $r_v(t) = \arg\min\{w(e)|e \notin T, t \in C(e), e \notin E(v)\}$ and sets $R(v) = \{r_v(t)|t \in T(v)\}$. However, this less naive algorithm does not correctly determine $R(v)$. For example, let $\delta_T(v) = 4$, with tree edges s, s', t, and t' incident on v, and let nontree edges 1, 2, and 3 be as in Figure 3. For this example, $r_v(s) = r_v(s') = 1$ and $r_v(t) = r_v(t') = 2$, so that $\{r_v(t)|t \in T(v)\} = \{1, 2\}$. However, $R(v) = \{1, 2, 3\}$, because three edges are needed to connect the components of $T - v$.

3.2.2 Transformation

FNR transforms G into a new graph G^N to closely simulate the greedy MST algorithm on each $CG(v)$. First, FNR replaces each node v with $\delta_T(v)$ nodes $v_1, \ldots, v_\delta$, where $\delta = \delta_T(v)$. These nodes have a one-to-one correspondence with the nodes in $CG(v)$, so that taking the *union* of v_i and v_j, for example, is equivalent to merging the ith and jth nodes of $CG(v)$. Second, FNR connects the new nodes with $\delta_T(v) - 1$ star edges (v_1, v_i), making v_1 the central node.

FNR uses a star instead of a clique for two reasons. Each star adds $O(\delta_T(v))$ edges, for a total of $O(n)$ edges; a clique would add $O((\delta_T(v))^2)$ edges, for a total of $O(n^2)$. In addition, a star has a natural tree structure: the central node v_1 is the "parent" of the star.

Still, FNR sometimes merges two nodes v_i and v_j, where $i > 1$ and $j > 1$. In this case, the *union* of v_i and v_j does not correspond to the predefined union tree. These non-tree *unions* of FNR contribute an $\alpha(n)$ increase in the time, where $\alpha(n)$ is the inverse of Ackermann's function.

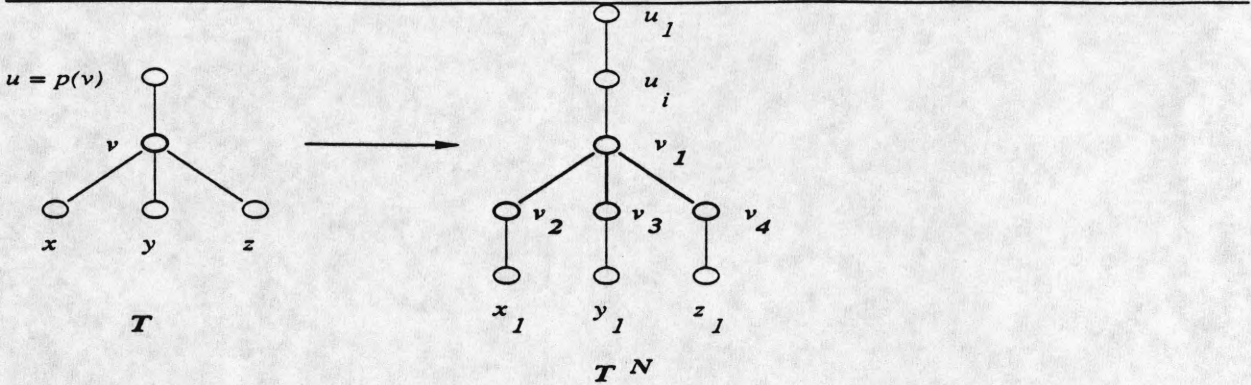

Figure 4: Transformation of node v with $\delta_T(v) = 4$ and $u = p(v)$

Let $G^N = (V^N, E^N)$ and let T^N be the MST of G^N. Figure 4 shows one node v that has been replaced by $\delta_T(v)$ nodes. These new nodes are called v's *star nodes.* Each *star edge* (v_1, v_i) in v's star is contained in T^N; we extend the parent notation from T to T^N so that $v_1 = p(v_i), i > 1$. T^N consists of these star edges plus edges corresponding to the edges of T: For each tree edge $t = (u, v)$ in T with $u = p(v)$, the corresponding edge in T^N is (u_i, v_1), for some i, with $u_i = p(v_1)$. E^N also contains edges corresponding to the nontree edges of $E - T$: For each nontree edge $e = (x, y)$ in $E - T$, the corresponding edge in E^N is (x_1, y_1). That is, all nontree edges are incident on the central node v_1 in each star. Finally, let $E^N(v)$ be a set of edges in G^N that are incident on one of $v_1, v_2, \ldots, v_\delta$.

3.2.3 Overview of FNR

After transforming G to G^N, FNR proceeds in stages similar to the stages in FER. In stage k of FNR, G_k^N is the analogue of G_k, and T_k^N is the MST of G_k^N. For the component graphs, let $CG_k(v)$ correspond to G_k^N. The supernodes in $CG_k(v)$ contain nodes of $CG(v)$ that have been merged in stages $1, \ldots, k-1$. Also, the edges $e_1, \ldots, e_{k-1}$ are not in $CG_k(v)$. Intuitively, for each v in the cycle of e_k, FNR determines whether e_k connects two different supernodes in $CG_k(v)$. Because FNR considers edges in order of increasing weight, e_k is the minimum weight edge connecting these two supernodes and is part of the MST of $CG(v)$. Therefore, the endpoints of e_k in $CG_k(v)$ are merged into one supernode in $CG_{k+1}(v)$. Otherwise, that is, when e_k is a self-loop in $CG_k(v)$, FNR does nothing during stage k, and $CG_{k+1}(v) = CG_k(v)$.

Formally, let $e_k = (u, v)$ in G^N and let (u', v') in G_k^N correspond to e_k. C_k^N is the cycle formed by adding (u', v') to T_k^N. Suppose that $l' = \text{lca}(u', v')$ in T_k^N. Basically, FNR adds e_k to $R(v)$ if node $v \in C_k^N$, except possibly for the nodes in u', v', and l'. By $v \in C_k^N$, we mean that at least one of v's star nodes v_i is the endpoint of an edge in G^N that corresponds to an edge in C_k^N.

Next, FNR contracts C_k^N. The basic operation is the *union* of two supernodes. The general case is the *union* of a supernode x' and $p(x')$, its parent in T_k^N. Let $t = (x_i, p(x_i))$ in T^N correspond to x', where x_i is in x's star. If t is a star edge, then the *union* of x' and $p(x')$ corresponds to the merge of two components in $CG(x)$ by the greedy MST algorithm. If t is not a star edge, then the *union* of x' and $p(x')$ corresponds to the contraction of an edge in T.

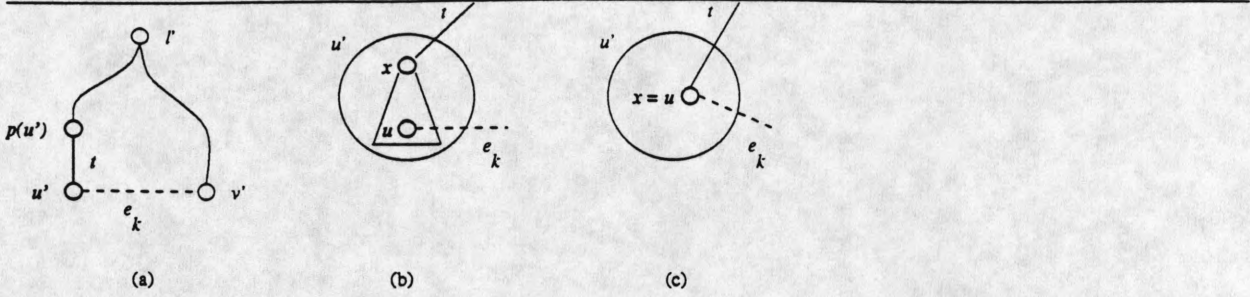

Figure 5: Cases for u', one endpoint of e_k: (a) the cycle C_k^N; (b) contents of u' when $x \neq u$; (c) contents of u' when $x = u$

Because e_k is incident on u and v in T^N, FNR has additional constraints for the *union* of u' and $p(u')$ and the *union* of v' and $p(v')$. In addition, FNR may have a special case for the *union* of two children of l'. The next section describes how to handle these cases.

The invariant for FNR formalizes the correspondence between G_k^N and $CG_k(v)$, for each v. The star nodes of v that are contained in supernode v' of G_k^N comprise one supernode in $CG_k(v)$. In other words, the *unions* taken in stages $1, \ldots, k-1$ correspond to the merges of supernodes in $CG_1(v), \ldots, CG_{k-1}(v)$. For example, suppose supernode v' in G_k^N contains $v_{i1}, \ldots, v_{il}$, and no other star nodes of v. The invariant ensures that $v_{i1}, \ldots, v_{il}$ are all contained in one supernode in $CG_k(v)$; furthermore, the invariant implies that no other star nodes of v are in that supernode in $CG_k(v)$.

The termination conditions for FNR are similar to the conditions for FER: after stage k, stop if T_{k+1}^N has only one supernode or if $k = m - n + 1$.

3.2.4 Endpoints of e_k and Special Case of *Union*

In this section we describe how FNR handles the endpoints of e_k and their lowest common ancestor. As in Section 3.2.3, $e_k = (u, v)$ corresponds to (u', v') in G_k^N. FNR checks whether it should take the *union* of u' and $p(u')$ and the *union* of v' and $p(v')$. Since the cases are similar to each other, we describe the case for u'.

To determine whether to take the *union* of u' and $p(u')$, FNR checks the tree edge adjacent to e_k. For u', let $t = (x, p(x))$ be the edge in T^N that corresponds to $(u', p(u'))$ in T_k^N, as in Figure 5(a). When $x \neq u$, as in Figure 5(b), e_k is not incident on x; as a result, FNR adds e_k to $R(x)$ and takes the *union* of u' and $p(u')$ in T_k^N. When $x = u$, as in Figure 5(c), FNR checks whether u is a leaf in T^N. If so, then $R^*(u) = \emptyset$, and although e_k is incident on u, FNR takes the *union* of u' and $p(u')$. Otherwise, when $x = u$ and x is not a leaf, FNR does not take the *union* of u' and $p(u')$ and does not add e_k to $R(v)$.

Also as in Section 3.2.3, let $l' = \text{lca}(u', v')$ in T_k^N. We assume that $u' \neq v'$; otherwise, $C_k^N = e_k$ and FNR proceeds to stage $k+1$. Without loss of generality, $l' \neq u'$. Define c_L to be the "left" child of l' in the path $u' \rightsquigarrow l'$ and let edge s in T^N be the edge corresponding to (c_L, l') in T_k^N. If $l' \neq v'$ as well, then define c_R to be the "right" child of l' in the path $v' \rightsquigarrow l'$ and let s' in T^N be the edge corresponding to (c_R, l'). Otherwise, when $l' = v'$, c_R and s' are defined to be `null`. Figure 6 illustrates this notation: Figure 6(a) shows $C^N(e_k)$ in the initial tree T^N; Figures 6(b) and (c) show two cases when $l' \neq v'$; and Figure 6(d)

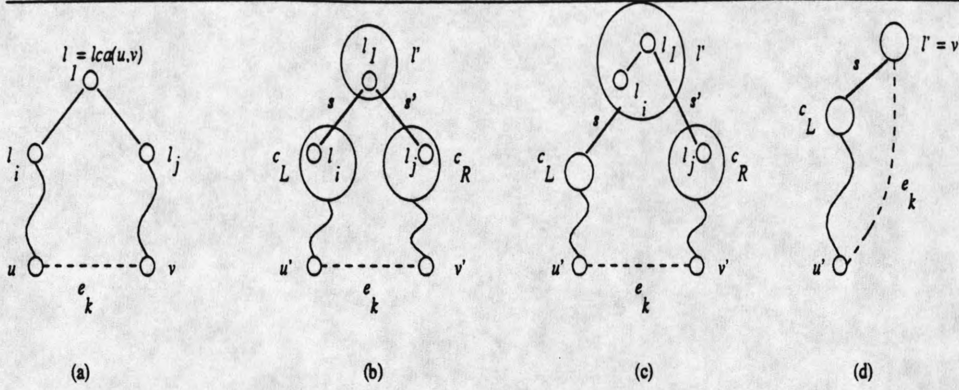

Figure 6: Special case for l': (a) $C^N(e_k)$; (b) s and s' in same star; (c) s and s' not in same star; (d) $s' =$ `null`

shows the case when $l' = v'$. In the figure, l_1 is the lowest common ancestor of u and v in T^N, and l_i and l_j are the star nodes in the paths $u \rightsquigarrow l_1$ and $v \rightsquigarrow l_1$, respectively.

The special case of *union* occurs when s and s' are in the same star, as in Figure 6(b). The edges s and s' are in the same star when l_1, l_i, and l_j are in different supernodes in G_k^N. In this case, FNR takes the *union* of c_L and c_R. Let l be the node in V corresponding to l_1, l_i, and l_j. In $CG_k(l)$, the supernodes containing l_i and l_j are merged.

Otherwise, s and s' are not in the same star, as in Figures 6(c) and (d). In Figure 6(d), $s' =$ `null`, so s and s' are trivially not in the same star. FNR takes the *union* of c_L and l' and of c_R and l'. Because l' is the parent of c_L and c_R, these *unions* are the general case of *union.*

3.2.5 Details of FNR

In each stage k, FNR decides whether to add e_k to $R(v)$ for each v in C_k^N. To keep track of which nodes have been considered, FNR uses the boolean variables $\rho(v_i, t)$, which are defined with respect to tree edges incident on v's star nodes. For each v_i in v's star, for each t in T^N incident on v, $\rho(v_i, t)$ is initially `false`. Let $\rho_k(v_i, t)$ be the value of $\rho(v_i, t)$ at the end of stage k. In general, during stage k, FNR changes $\rho(v_i, t)$ to `true` if t corresponds to an edge in C_k^N, e_k is not in $E_N(v)$, and $\rho_{k-1}(v_i, t) =$ `false`.

For a leaf node z in T, $R^*(z) = \emptyset$. In this case, for the corresponding leaf z_1 in T^N, FNR initially sets $\rho(z_1, t) =$ `true`, where t is the one tree edge in T^N incident on z_1. As a result, FNR trivially determines that $R(z) = \emptyset$.

During stage k, FNR contracts C_k^N by taking the *union* of supernodes along the cycle. Let e_k correspond to (u', v') in C_k^N and let $l' = \text{lca}(u', v')$ in T_k^N, as in Section 3.2.3. The general case of a *union* consists of the *union* of a supernode in T_k^N and its parent. Let x' and $p(x')$ be these supernodes, and let $t = (x_i, p(x_i))$ in T^N correspond to $(x', p(x'))$. FNR checks whether $\rho(x_i, t) = \rho(p(x_i), t) =$ `true`; if so, then FNR takes the *union* of x' and $p(x')$.

Let c_L, c_R, s, and s' be as defined in Section 3.2.4. If s and s' are not in the same star, then the *union* of l' and c_L and the *union* of l' and c_R fall under the general case of *union.* Otherwise, s and s' are in the same star, with l_1, l_i, and l_j in different supernodes of G_k^N, as in Figure 6(b). In this case, to record that FNR takes the *union* of c_L and c_R, FNR changes

$\rho(l_i, s)$ and $\rho(l_j, s')$ to `true`, but $\rho(l_1, s)$ and $\rho(l_1, s')$ remain `false`. In addition, after the *union* of c_L and c_R, two tree edges connect l' and the new supernode. One of these edges is removed to ensure that T_{k+1}^N is a tree, instead of a multigraph.

Figure 7 shows that pseudocode of FNR.

3.2.6 Example of FNR

This section illustrates how FNR operates and how FNR handles its special cases. Let G be the graph shown in Figure 8(a), with the solid lines representing its MST T. The nontree edges e_1, e_2, e_3, and e_4 are represented by dashed lines and are labeled by their weights, $1, 2, 3$, and 4, respectively.

Figure 8(b) shows T^N. Node b has been replaced by a four-node star and node c has been replaced by a three-node star. For the first stage, $T_1^N = T^N$, and cycle C_1^N passes through d_1, b_3, b_1, b_4, and f_1. In this stage, $l' = b_1$, edge $s = (b_1, b_3)$, and $s' = (b_1, b_4)$. Because s and s' are in the same star in T^N, FNR takes the *union* of b_3 and b_4. Hence, s and s' are *not* contracted, but s' is deleted (otherwise, T_2^N would have two tree edges from b_1 to $b_3b_4d_1f_1$). On the other hand, both d_1 and f_1 are leaves in T_1^N, so (b_3, d_1) and (b_4, f_1) are contracted.

The resulting tree T_2^N is shown in Figure 9. In stage 2, $s = (a_1, b_1)$ and $s' = \texttt{null}$. Hence, s and s' are not in the same star in T^N, so s is contracted. In addition, because e_2 is in $E^N(c)$, FNR does *not* take the *union* of c_1 and b_1. As a result, T_3^N has $a_1b_1b_2$ as a supernode instead of $a_1b_1b_2c_1$.

In stage 3, as shown in Figure 10(a), the edge in G_3^N corresponding to $e_3 = (b_1, h_1)$ is incident on supernode $a_1b_1b_2$ in T_3^N. In the cycle C_3^N, the edge $(a_1b_1b_2, c_1)$ corresponds to $t = (b_2, c_1)$ in T^N. Therefore, e_3 is not adjacent to t in T^N, and FNR contracts t. After all the tree edges in C_3^N are processed, the resulting tree T_4^N is as shown in Figure 10(b). In a similar manner, the edge $(a_1b_1b_2c_1c_3h_1, b_3b_4d_1f_1)$ corresponds to (b_1, b_3) in T^N, which is not adjacent to edge $e_4 = (d_1, g_1)$ in T^N; therefore, FNR contracts $(a_1b_1b_2c_1c_3h_1, b_3b_4d_1f_1)$.

After stage 4, T_5^N consists of one supernode, hence FNR terminates. (In addition, $k = m - n + 1$.) At this point, $R(a) = R(d) = R(f) = R(g) = R(h) = \emptyset$, because all these nodes are leaves in T. $R(b) = \{e_1, e_2, e_4\}$ and $R(c) = \{e_3, e_4\}$.

3.2.7 Correctness of FNR

First, we review Chin and Houck's result [CH 78]. They prove that the set of edges in the MST of $CG(v)$ is the correct set of edges to replace v. Let $T(\overline{v})$ be the set of edges in $T - v$.

Lemma 1 *[CH 78, Lemma 2, p. 334] If $R^*(v)$ is the set of edges in the minimum spanning tree of $CG(v)$, then $T(\overline{v}) \cup R^*(v)$ is the set of edges in the minimum spanning tree of $G - v$.*

As a result of Lemma 1, to show that FNR is correct, we only need to prove that $R(v)$ is the set of edges in the MST of $CG(v)$ for each v, that is, $R(v) = R^*(v)$. As an intermediate step, we show that the FNR invariant correctly makes the relationship between $CG(v)$ and G^N explicit.

Lemma 2 (FNR Invariant) *Nodes $v_{i1}, \ldots, v_{il}$ are the only star nodes of v in supernode u' of G_k^N if and only if $v_{i1}, \ldots, v_{il}$ comprise one supernode of $CG_k(v)$.*

Preprocessing
Replace each v in V with star

Stage k
/* Derive T_{k+1}^N from T_k^N */
If endpoints of e_k are in same supernode in T_k^N, then
  $T_{k+1}^N := T_k^N$
Else
  Determine C_k^N by finding l'

  /* General case of *union* */
  For each $(x', p(x'))$ in $C_k^N \cap T_k^N$,
      except edges corresponding to s and s'
    Let $t = (x_i, p(x_i))$ in T^N correspond to $(x', p(x'))$
      and let x_i be in x's star and $p(x_i)$ be in y's star
    If $\rho(x_i, t) = \texttt{false}$ and $e_k \notin E^N(x)$, then
      Add e_k to $R(x)$
      Set $\rho(x_i, t) := \texttt{true}$
    If $\rho(p(x_i), t) = \texttt{false}$ and $e_k \notin E^N(y)$, then
      Add e_k to $R(y)$
      Set $\rho(p(x_i), t) := \texttt{true}$
    If $\rho(x_i, t) = \rho(p(x_i), t) = \texttt{true}$, then
      Take *union* of x' and $p(x')$

  /* Special case for l' */
  If s and s' are in same star in T^N, e.g., l's star, then
    Take *union* of c_L and c_R
    Add e_k to $R(l)$
    Remove (c_R, l') from T_{k+1}^N
  Else
    Take *union* of c_L and l' and *union* of c_R and l',
      similarly to case of $(x', p(x'))$ above

Figure 7: Pseudocode for FNR

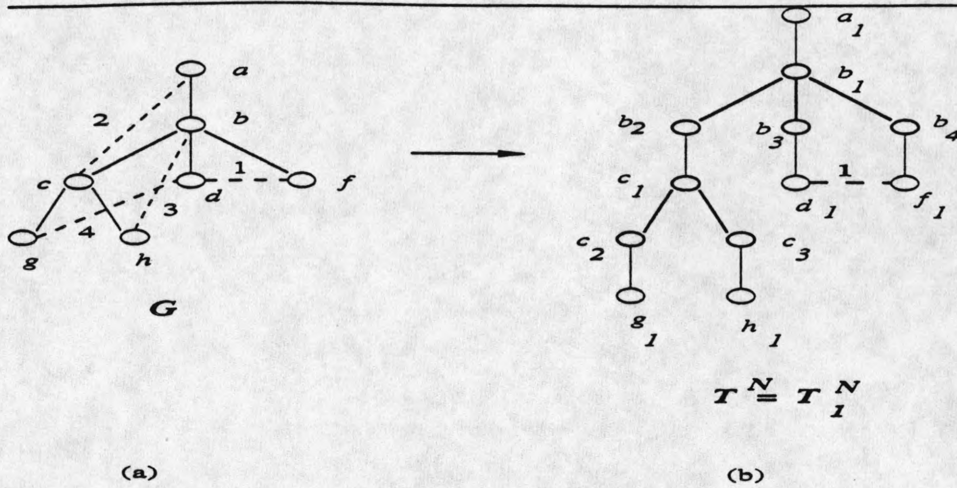

Figure 8: Example graph for FNR, before and after transformation

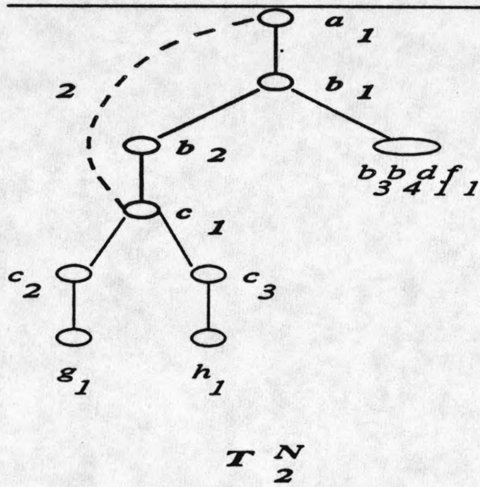

Figure 9: Example for FNR, second stage

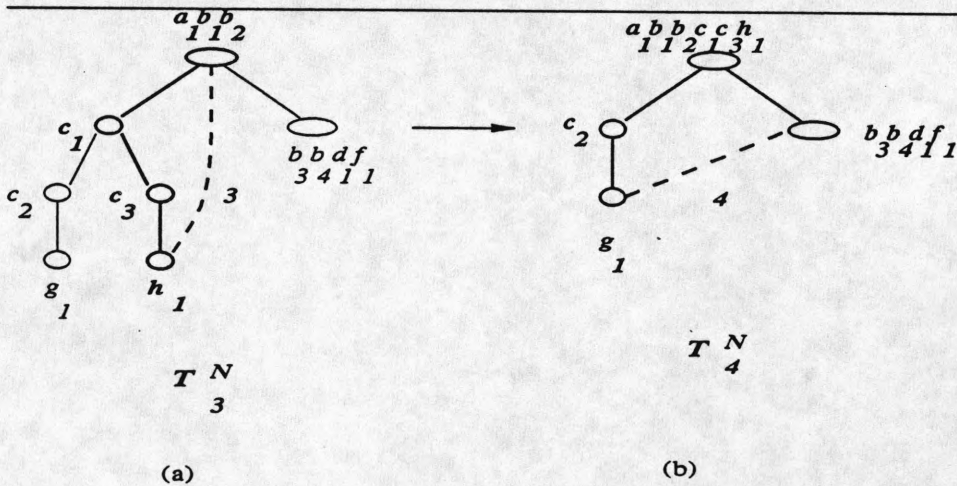

Figure 10: Example for FNR, third and fourth stages

Proof: Fix node v in V. We prove the lemma by induction on k. The base case is $k = 1$: $CG_1(v) = CG(v)$ and $G_1^N = G^N$. Initially, there is a one-to-one correspondence between the star nodes of v and the nodes of $CG(v)$, so the lemma is true for $k = 1$.

For the inductive step, assume that the inductive hypothesis holds for k. We show that the lemma holds in one direction for $k+1$: if $v_{i1}, \ldots, v_{il}$ are the only star nodes of v in supernode u' of G_{k+1}^N, then $v_{i1}, \ldots, v_{il}$ comprise one supernode of $CG_{k+1}(v)$. The opposite direction is easy to prove, and the details are left to the reader.

The proof of the inductive step considers several cases for e_k and its relation to v in G_k^N. In general, each case leads to one of two possibilities in $CG_k(v)$: either e_k is a self-loop in $CG_k(v)$, or e_k connects two different supernodes in $CG_k(v)$. If e_k is a self-loop in $CG_k(v)$, then $CG_{k+1}(v) = CG_k(v)$. If e_k connects two different supernodes in $CG_k(v)$, then there is one supernode in $CG_{k+1}(v)$ that contains the star nodes in the original two supernodes. The case analysis of the rest of the proof shows that the same change occurs in G_k^N.

Kruskal's algorithm and FNR consider one nontree edge at a time. Suppose e_k is a self-loop in G_k^N. Consequently, by the procedure of FNR, $G_{k+1}^N = G_k^N$. There are two cases for e_k corresponding to $CG_k(v)$: either e_k is incident on a star node of v, so that $e_k \notin CG_k(v)$; or e_k is not incident on a star node of v, so that by the inductive hypothesis, e_k is a self-loop in $CG_k(v)$. In either case, $CG_{k+1}(v) = CG_k(v)$. Therefore, for each supernode u' of G_{k+1}^N, the star nodes of v that are contained in u' comprise one supernode of $CG_{k+1}(v)$.

Suppose that e_k is not a self-loop in G_k^N. We say that node v is in C_k^N if one of v's star nodes is contained in a supernode in C_k^N. Also, by $v_1 = lca(e_k)$, we mean that v_1 is the lowest common ancestor of the endpoints of e_k in T^N. We consider three subcases: v is not in C_k^N; v is in C_k^N and $v_1 \neq \mathrm{lca}(e_k)$; and v is in C_k^N and $v_1 = \mathrm{lca}(e_k)$. The second subcase corresponds to the general case of a *union*, and the third subcase corresponds to the special case relating to l'.

When v is not in C_k^N, the supernodes in G_k^N that contain star nodes of v are not changed during stage k. Likewise, e_k is a self-loop in $CG_k(v)$, so that $CG_{k+1}(v) = CG_k(v)$.

When v is in C_k^N and $v_1 \neq \mathrm{lca}(e_k)$, only one star edge of v, (v_1, v_i), might correspond to an edge in C_k^N. If v_1 and v_i are in the same supernode in G_k^N, this supernode is not affected during stage k of FNR. Likewise, by the inductive hypothesis, e_k is a self-loop in $CG_k(v)$, so that $CG_{k+1}(v) = CG_k(v)$. On the other hand, if v_1 and v_i are in different supernodes in G_k^N, say u' and v', then FNR takes the *union* of u' and v' in stage k. Likewise, by the inductive hypothesis, e_k connects two different supernodes in $CG_k(v)$ that are merged into one supernode in $CG_{k+1}(v)$, preserving the inductive hypothesis.

Finally, when v is in C_k^N and $v_1 = \mathrm{lca}(e_k)$, two star edges of v, say (v_1, v_i) and (v_1, v_j), might correspond to edges in C_k^N. If v_1, v_i, and v_j are each in different supernodes in G_k^N, then FNR takes the *union* of the supernodes containing v_i and v_j. Note that the endpoints of e_k are descendants of v_i and v_j, respectively. Therefore, by the inductive hypothesis, in $CG_k(v)$, e_k connects the supernodes containing v_i and v_j, and these two supernodes are merged into one supernode of $CG_{k+1}(v)$. If one, but not both, node of v_i and v_j is in the same supernode of G_k^N as v_1, then FNR takes the *union* of v_1's supernode and the other supernode. Likewise, a corresponding merge occurs between supernodes of $CG_k(v)$. Finally, if v_1, v_i, and v_j are all in the same supernode in G_k^N, then by the inductive hypothesis, e_k is a self-loop in $CG_k(v)$. In both situations, the supernodes containing v's star nodes are not affected. □

As a corollary of Lemma 2, FNR correctly simulates Kruskal's algorithm on each component graph. Therefore, FNR determines the set of edges in the MST of each $CG(v)$.

Theorem 1 *FNR determines the correct node replacement for each node v in V.*

3.2.8 Time Analysis for FNR

For FNR, the transformation from G to G^N takes $O(m)$ time total: $O(\sum_v \delta_T(v)) = O(n)$ time for the new star nodes and edges, and $O(m)$ time for the new endpoints of nontree edges.

As with FER, the number of operations to contract each C_k^N is proportional to $|C_k^N \cap T_k^N|$. In addition, in each stage k where the endpoints of e_k are not in the same supernode in T_k^N, at most four edges in $u' \rightsquigarrow l'$ and $v' \rightsquigarrow l'$ are not contracted. In total, $\sum_k |C_k^N \cap T_k^N| = |T^N|$; because only $O(n)$ star edges are added in the transformation from T to T^N, $|T^N| \leq 2|T|$. Hence, the number of operations to determine and contract the cycles is

$$\sum_k |C_k^N \cap T_k^N| \leq |T^N| + \sum_{|C_k^N \cap T_k^N|>0} 4 = 5|T^N| = O(n).$$

Because some *unions* might not be in the predefined union tree, the time for these operations is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann's function.

Finally, FNR takes $O(m)$ time to check the endpoints of each e_k. Therefore, the total time for FNR is $O(m + n\alpha(n))$.

4 FERP–Find Edge Replacements in Parallel

The algorithms FERP is the parallel analogue of FER. FERP takes $O(\log n)$ time, using m processors on a CREW PRAM. This section provides the motivation for the structure of FERP by first describing a similarity between the sequential algorithm FER and ear decomposition. Next, an overview of FERP is given, followed by the details of the tree contraction step used in FERP and a method of a compact representation for values used in the tree contraction step.

4.1 FER and Ear Decomposition

The sequential algorithm FER determines and contracts the cycles $C_1, C_2, \ldots,$ in turn, one after the other. The parallel algorithm FERP cannot afford to do so; otherwise, the running time of FERP could be $\Omega(m)$, the same as for FER. The challenge for FERP is to determine and contract the cycles in parallel, that is, at the same time. To meet that challenge, we consider the similarity between the outputs of FER and of one particular method of ear decomposition.

FER partitions the tree edges of E into subsets indexed by the nontree edges. Specifically, the subsets of the partition are $F(e) = \{t | r(t) = e\}$. $F(e)$ is a subset of the cycle $C(e)$. Likewise, the parallel ear decomposition algorithm of Maon *et al.* [MSV 86] partitions the tree edges of E into subsets indexed by the nontree edges. In this latter algorithm, the

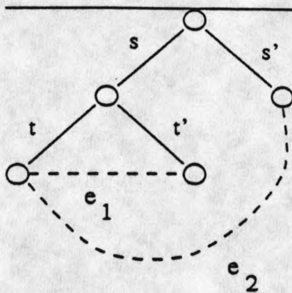

Figure 11: Contrast between FER and ear decomposition: t is assigned to $F(e_1)$ and to $P(e_2)$, not $P(e_1)$. $F(e_1)$ consists of edges t and t', and $P(e_2)$ consists of edges t, s, and s'.

subsets are called *ears*; the ear $P(e)$ (to be defined in the next paragraph) is also a subset of the cycle $C(e)$.

On the other hand, the two algorithms use different partition rules. Let the *level* of nontree edge e be the level in T of the lowest common ancestor of the endpoints of e, where the root of T is at level 0. FER assigns tree edge t to $F(e)$ if e is the minimum weight nontree edge such that $t \in C(e)$. The algorithm of Maon *et al.* assigns t to $P(e)$ if e is the minimum *level* nontree edge such that $t \in C(e)$. For example, in Figure 11, $t \in C(e_1)$ and $t \in C(e_2)$. Suppose that $w(e_1) < w(e_2)$ and $\text{level}(e_1) > \text{level}(e_2)$. In this case, FER assigns t to $F(e_1)$, but the algorithm of Maon *et al.* assigns t to $P(e_2)$.

In summary, the similarity between FER and the algorithm of Maon *et al.* suggests that the basic structure of FERP should be be similar to the structure of the algorithm of Maon *et al.*, which is a *parallel* algorithm that takes time $O(\log n)$, using m processors, on a CREW PRAM.

4.2 Details of FERP

The major steps of FERP follow the basic structure of the algorithm of Maon *et al.* The main difference is in the tree contraction step. The steps of FERP are as follows:

1. Root T and compute the parent and level of each node
2. Sort the nontree edges by weight
3. Determine the level of each nontree edge
4. Use tree contraction to determine $r(t)$ for each tree edge $t = (u, v)$, where u is the parent of v
   (a) Let $f(u, i) = \min\{w(e) | e = (u, v) \notin T, \text{level}(e) \leq i\}$, for $0 \leq i \leq \text{level}(u)$. $f(u, i)$ is the minimum weight of a nontree edge incident on u such that the lca of the edge's cycle is at or above level i in T.
   (b) $r(t)$ is the edge whose weight is $w^*(t) = \min\{f(x, \text{level}(u)) | x \in T_v\}$, where T_v is the subtree of T rooted at v. The conditions ensure that $r(t)$ is the nontree edge of minimum weight such that one endpoint is in v's subtree and the lca of $r(t)$'s endpoints is an ancestor of u, that is, t is in $C(r(t))$.

FERP uses well known algorithms to carry out the all the steps. To root T and compute the parents and levels of nodes in T, FERP uses an Euler-tour technique (see for example [Ja 92, pp. 108–118]). To sort the nontree edges by weight, FERP uses Cole's parallel merge-sort algorithm [Co 88]. Last, to determine the levels of the nontree edges, FERP uses a parallel lowest common ancestors algorithm, such as the algorithm of Schieber and Vishkin [SV 88]. Each of these steps takes $O(\log n)$ time. The sorting requires m processors, while the other steps use only n processors, on a CREW PRAM.

For the final step, to set $r(t)$ for each tree edge t, FERP evaluates the expression $w^*(t)$ by tree contraction. This tree contraction step takes $O(\log n)$ time, using m processors, on a CREW PRAM. The next section explains in more detail how FERP does the tree contraction.

In summary, FERP consists of a finite number of steps that each take $O(\log n)$ time, useing either n or m processors on a CREW PRAM. Hence, FERP takes $O(\log n)$ time, using m processors, on a CREW PRAM.

4.3 Tree Contraction to Set Edge Replacements

FERP uses the tree contraction algorithm of [RMM 93, pp. 187–190] to evaluate the expression $w^*(t)$ for each t in T. Normally, when each node in the tree has one input value, the tree contraction takes $O(\log n)$ time, using $n/\log n$ processors on a CREW PRAM; in FERP's case, where each node u has a vector $f(u,i)$ of inputs, the tree contraction takes m processors. This algorithm runs on trees of unbounded degree, provided two conditions are satisfied: the *rake* operation takes at most $O(\log c)$ time when applied to a node with c children that are leaves; and the rake and *compress* operations can be applied to different parts of the tree asynchronously. In this section, we show these two conditions are satisfied.

First, we define a function $h(u,i)$ on the nodes in T. For each node u, $h(u,i)$ is the minimum of the $f(v,i)$ values in T_u, the subtree of T rooted at u. That is,

$$h(u,i) = \min\{f(v,i) | v \in T_u\}.$$

Let $t = (u,v)$ be an edge in T, where $u = p(v)$. The replacement edge $r(t)$ connects the two components of $T - v$; hence, one endpoint of $r(t)$ is in the subtree of v and the level of $r(t)$ is at most level(u), as shown in Figure 12. In other words, $r(t)$ is the edge with weight $h(v, \text{level}(u)) = h(v, \text{level}(v) - 1)$. Therefore, the tree contraction algorithm actually computes $h(v, \text{level}(v) - 1)$ for each v in V.

For each internal node u in T, let $D(u)$ be the set of u's children. The tree contraction algorithm maintains the following invariant:

$$h(u,i) = \min\{f(u,i), \min\{\min(b(v,i), h(v,i)) | v \in D(u)\}\}, \tag{1}$$

where $b(v,i)$ is called the *label* of v for level i. The value of $b(v,i)$ is adjusted throughout the computation to maintain the invariant; initially, $b(v,i) = f(v,i)$. The value of $\min(b(v), h(v,i))$ is called the level i *contribution* of v to its parent u.

The rake operation eliminates all the leaves while maintaining the invariant (1). Let T^k be the tree after the kth rake and suppose y is a leaf in T^k. Since y is the only node in

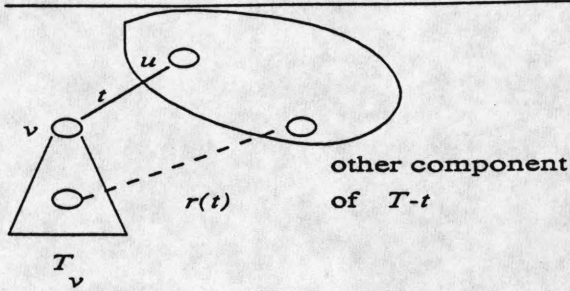

Figure 12: The correspondence between $r(t)$ and $h(v, \text{level}(u))$: to connect $T-t$, one endpoint of $r(t)$ is in T_v and $\text{level}(r(t)) \leq \text{level}(u)$.

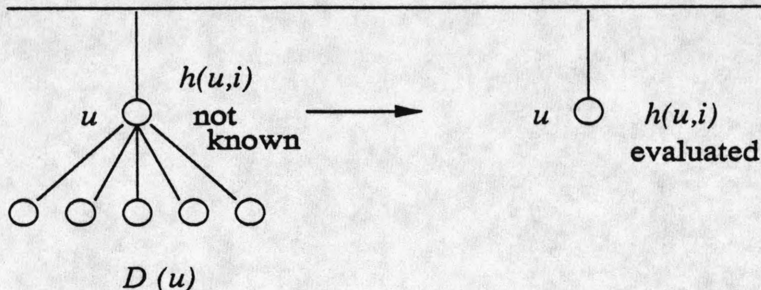

Figure 13: A rake applied to node u. Before the rake, $h(u,i)$ is not known, and after $h(u,i)$ has been evaluated.

T_y^k, $h(y,i) = f(y,i)$ for all i. Consider a node u such that all its children are leaves. The invariant (1) reduces to

$$h(u,i) = \min\{f(u,i), \min\{\min(b(v,i), f(v,i))|v \in D(u)\}\}. \tag{2}$$

A rake applied to u removes all the leaves in $D(u)$ and evaluates $h(u,i)$, as shown in Figure 13. We show in Section 4.4 how FERP can look up the value of $f(u,i)$ in $O(1)$ time, using m processors in total for each rake. Similarly, FERP looks up the values of $b(v,i)$, which also takes $O(1)$ time; this step uses at most m processors for one rake. Afterwards, Equation (2) computes the minimum of $2|D(u)|+1$ values, which takes $O(\log|D(u)|)$ time, using $|D(u)|$ processors. Therefore, the rake applied to u takes $O(\log|D(u)|)$ time, which satisfies the first condition of the tree contraction algorithm.

The compress operation shortens each *chain* in the current tree. A chain in a tree is a set of nodes such that each node has only one child. Suppose that u, v, and x form a chain such that $u = p(v)$ and $v = p(x)$. Then a compress applied to this chain removes v from the tree and sets $u = p(x)$. Figure 14 shows the effect of this compress. Let $b'(x,i)$ be the value of x's label after the compress. By the invariant (1), we have

$$\begin{aligned} h(u,i) &= \min\{f(u,i), \min(b(v,i), h(v,i))\} \\ &= \min\{f(u,i), \min(b(v,i), \min\{f(v,i), \min(b(x,i), h(x,i))\})\} \\ &= \min\{f(u,i), \min(\min(b(v,i), f(v,i), b(x,i)), h(x,i))\} \\ &= \min\{f(u,i), \min(b'(x,i), h(x,i))\}. \end{aligned}$$

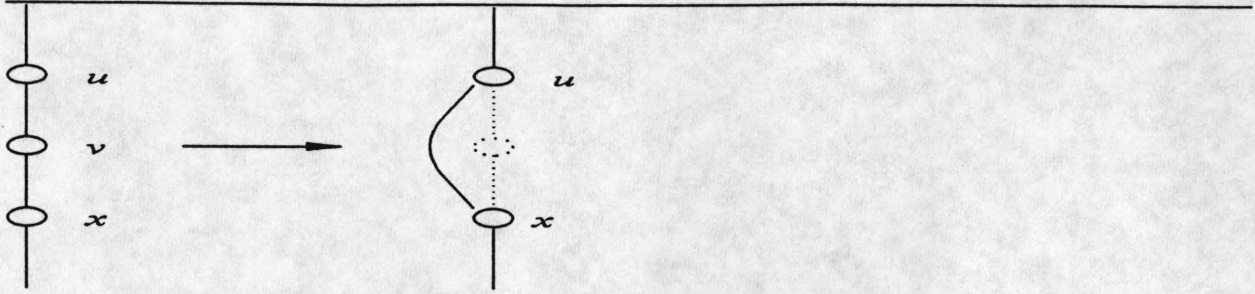

Figure 14: A compress applied to the chain u, v, and x.

That is, $b'(x,i) = \min((b(v,i), f(v,i), b(x,i))$. The update to x's label requires no $b(\cdot,i)$, $f(\cdot,i)$, or $h(\cdot,i)$ values from x's strict descendants. Therefore, this compress is independent of a rake applied to different parts of the tree. Hence, the second condition of the tree contraction algorithm is satisfied.

The next section shows how to look up the $f(u,i)$ and $b(u,i)$ values in $O(1)$ time, using only m processors.

4.4 Computing $f(u,i)$ and $b(u,i)$

For the FERP tree contraction step, the $f(u,i)$ and $b(u,i)$ values could be stored in a lookup table of size $\Omega(n \times d)$, where d is the height of T. In the worst case, $d = \Omega(n)$, so that the size of the table for f and b would be $\Omega(n^2)$. To ensure a constant lookup time, FERP would need $\Omega(n^2)$ processors, which is more than we have available. Instead, we use a compact representation for the $f(u,i)$ and $b(u,i)$ values that requires only m processors for a constant lookup time.

First, we discuss how to represent the $f(u,i)$ values. Let $\delta_E(u)$ be the nontree degree of node u in graph G, that is, the number of edges in $E - T$ incident on u. The key idea is that we represent $\delta_E(u)$ values for each node u, because there are only $\delta_E(u)$ nontree edges incident on u. The portion of the compact represenation for u has $\delta_E(u)$ entries; each entry has two components, a level number and an edge weight. For example, suppose that u has exactly three incident nontree edges at levels i_1, i_2, and i_3, where $i_1 < i_2 < i_3$. For $i < i_1$, $f(u,i)$ is undefined, because u has no incident nontree edges with a level less than i_1. Furthermore, for $i_1 \leq i < i_2$, $f(u,i) = f(u,i_1)$; similarly, for $i_2 \leq i < i_3$, $f(u,i) = f(u,i_2)$ and for $i_3 \leq i \leq \text{level}(u)$, $f(u,i) = f(u,i_3)$. As a result, to represent all $f(u,i)$ values, for $0 \leq i \leq \text{level}(u)$, we only need the entries $(i_1, f(u,i_1))$, $(i_2, f(u,i_2))$, and $(i_3, f(u,i_3))$. For all nodes in total, the compact lookup table uses $\sum_{u \in V} \delta_E(u) \leq 2m$ entries.

To determine $f(u,i)$ given u and i, we use $\delta_E(u)$ processors, one for each entry in u's portion of the lookup table. Each processor looks at the level numbers for its entry and its neighbor's entry, to check whether i is between these two level numbers. The processor that has i in its range of levels then returns the f value in its corresponding entry. For the example of the previous paragraph, suppose that $i_2 < i < i_3$. The processor for i_1 looks up i_1 and i_2 and does nothing since $i > i_2$; similarly, the processor for i_3 does nothing because $i < i_3$. The processor for i_2 looks up i_2 and i_3, notes that i is in this range, and returns $f(u,i_2)$.

Now, we show that the compact representation can be set up in $O(\log n)$ time, using m processors, on a CREW PRAM. Each node u has $\delta_E(u)$ processors available. First, we sort the edges incident on u lexicographically by level and then weight. Next, for each level, we determine the minimum weight nontree edge that is incident on u; each processor asks itself, "Is my corresponding edge weight the minimum of the level that I'm in?" For the processors that respond affirmatively, we use list ranking to order these processors by level and call this list the *compact list* for u. At this point, the entry for each processor in the compact list consists of its level number and the minimum weight of edges at that specific level; $f(u,i)$ is the minimum over edges at levels less than or equal to i. So, finally, for the levels $i_1, i_2, \ldots$ represented in the compact list for node u, we use a parallel prefix algorithm to compute $f(u,i_1), f(u,i_2), \ldots$.

To determine the total number of processors required for looking up $f(u,i)$ values in the FERP tree contraction step, we consider when $f(u,i)$ values are needed. At most one rake and one compress occur simultaneously. In a rake operation, from Equation (2), FERP looks up $f(u,i)$ and $f(v,i)$ for each v in $D(u)$. In a compress operation, FERP looks up $f(v,i)$ for each node v removed. In each stage, the rake and compress operate on different parts of the tree; therefore, for each node u, FERP looks up $f(u,i)$ at most once. In total, the number of processors required is at most $\sum_{u \in V} \delta_E(u) = 2m$.

Finally, we reduce the number of processors from $2m$ to m by having each processor simulate the two processors assigned to the endpoints of a nontree edge.

For the $b(u,i)$ values, the compact representation also requires only m processors for constant time lookup. Initially, $b(u,i_x) = f(u,i_x)$, where u has an incident edge with level i_x. Hence, node u has $\delta_E(u)$ processors for these $b(u,i_x)$ values, one per level i_x.

After a rake operation, suppose one of the nodes v in $D(u)$ has an incident edge with level i_y, but u does not. Initially, $b(u,i_y)$ has not been set and u has no processor assigned to $b(u,i_y)$. However, after the rake, v's processors are available; therefore, v's processor for level i_y becomes u's processor for level i_y. A similar reassignment can be done for a compress operation. Therefore, in total, at most m processors are needed to look up the $b(u,i)$ values.

5 FNRP–Find Node Replacements in Parallel

5.1 Overview of FNRP

Like FNR, FNRP simulates a greedy MST algorithm on each component graph simultaneously. Intuitively, FNRP simulates Sollin's algorithm [BGH 65], so that in each stage, FNRP determines the minimum weight outgoing edge from each connected component in $CG(v)$. In this manner, the number of distinct components in each $CG(v)$ is reduced by at least half after each iteration. Therefore, FNRP takes $O(\log \Delta)$ iterations, where Δ is the maximum degree of a node in G.

In each iteration, FNRP runs a modified FERP to determine the minimum weight outgoing edge from each component. The modified FERP and the merging of components require at most $O(\log n)$ time per iteration. Therefore, FNRP takes $O(\log n \log \Delta) = O(\log^2 n)$ time using m processors on a CREW PRAM.

5.2 Simulating Sollin's Algorithm

Initially, FNRP transforms G into G^N via the FNR transformation of Section 3.2.2. Then, in iteration k, FNRP finds an edge $r(v_i^{(k)})$ for each supernode $v_i^{(k)}$ and merges $v_i^{(k)}$ with supernode $s(v_i^{(k)})$, where $r(v_i^{(k)})$ and $s(v_i^{(k)})$ are defined in the following paragraphs.

For each iteration k, let $G^{(k)}$ be the graph at the beginning of the iteration and $T^{(k)}$ be its MST. $G^{(k+1)}$ is derived from $G^{(k)}$, and $T^{(k+1)}$ is derived from $T^{(k)}$, as described in this section. (The progression is similar to G_k^N and T_k^N of FNR, but those graphs and trees have different structures.) For $k = 1$, since $G^{(1)} = G^N$, we sometimes drop the $^{(1)}$ superscript, if the context is clear.

We use $v_i^{(k)}$ to denote a supernode in $G^{(k)}$. Each $v_i^{(k)}$ contains only nodes from v's star. To indicate which v_i are in $v_i^{(k)}$, we use the lowest numbered star node as $v_i^{(k)}$'s representative. For example, suppose $v_i^{(k)}$ comprises v_2 and v_5; we write $\text{rep}^{(k)}(v_2) = \text{rep}^{(k)}(v_5) = v_2$.

Corresponding to $G^{(k)}$, we define $CG^{(k)}(v)$ to be the component graph for v at the beginning of the kth iteration: each supernode in $CG^{(k)}(v)$ corresponds to one connected component of $CG(v)$. As an invariant, in each iteration of FNRP, the star nodes of v that comprise one supernode of $G^{(k)}$ comprise one supernode of $CG^{(k)}(v)$. As a result, we sometimes refer to $v_i^{(k)}$ both as a supernode in $CG^{(k)}(v)$ and as a supernode in $G^{(k)}$. Similarly, we sometimes refer to edge e in $G^{(k)}$ as its corresponding edge in $CG^{(k)}(v)$. We prove in Section 5.4 that $r(v_i^{(k)})$ corresponds to the minimum weight outgoing edge from $v_i^{(k)}$ in $CG^{(k)}(v)$.

Edge $r(v_i^{(k)})$ is defined as

$$r(v_i^{(k)}) = \arg\min\{w(e) | e \in E^N - T^N - E^{(k)}(v), (v_i^{(k)}, p(v_i^{(k)})) \in C^{(k)}(e)\},$$

where $p(v_i^{(k)})$ is the parent in $T^{(k)}$ of $v_i^{(k)}$. To be precise, the second constraint should be "the edge in T^N corresponding to $(v_i^{(k)}, p(v_i^{(k)}))$ is contained in $C^{(k)}(e)$," but we use the abbreviated form for ease of presentation.

The edge in $CG^{(k)}(v)$ corresponding to $r(v_i^{(k)})$ connects $v_i^{(k)}$ with another supernode. We define $s(v_i^{(k)})$ to be that supernode. Figure 15 shows examples of $s(v_i^{(k)})$. Section 5.3 explains how FNRP determines $s(v_i^{(k)})$. After determining $s(v_i^{(k)})$, FNRP merges $v_i^{(k)}$ and $s(v_i^{(k)})$. As shown in Figure 16, the merges may occur along a chain of supernodes, so that a supernode in $G^{(k+1)}$ may contain more than two supernodes in $G^{(k)}$. Without loss of generality, the supernodes in each star of $G^{(k+1)}$ are numbered in increasing order of the supernodes' representatives. Hence, $v_1^{(k+1)}$ contains $v_1^{(k)}$.

To preserve the tree structure of $T^{(k+1)}$, FNRP removes all but one star edge incident on each new supernode $v_i^{(k+1)}$, when $i > 1$. Suppose $v_i^{(k)}$ and $v_j^{(k)}$ are merged into $v_l^{(k+1)}$, and that $i < j$. FNRP removes the star edge corresponding to $(v_j^{(k)}, p(v_j^{(k)}))$. Consequently, $T^{(k+1)}$ has neither redundant tree edges nor self-loops.

FNRP also updates the $\text{rep}^{(k+1)}(v_i)$ values as it merges supernodes. Continuing the example, suppose $v_i^{(k)}$ has v_3 as its representative and $v_j^{(k)}$ has v_5 as its representative. In this case, $\text{rep}^{(k+1)}(v_i) = \min(v_3, v_5) = v_3$ for each v_i in $v_l^{(k+1)}$.

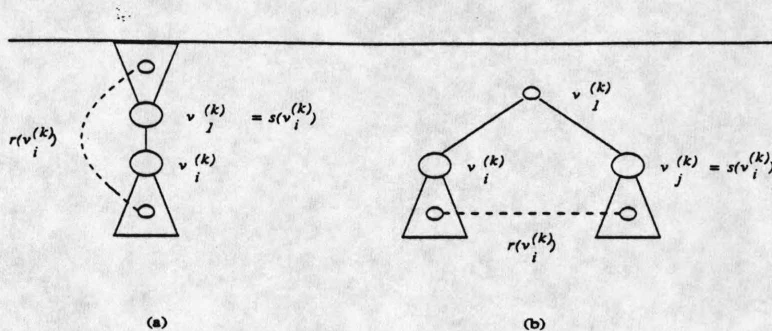

Figure 15: Two examples of $s(v_i^{(k)})$: (a) $s(v_i^{(k)}) = v_1^{(k)}$; (b) $s(v_i^{(k)}) = v_j^{(k)}, j > 1$, a supernode other than $v_1^{(k)}$.

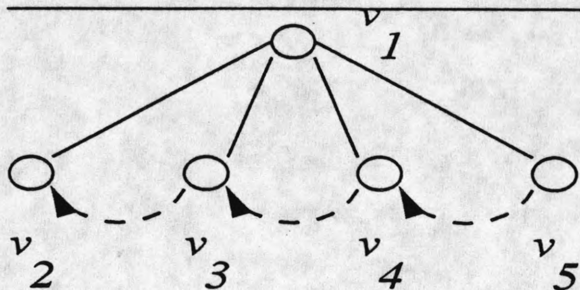

Figure 16: Merge along a chain of nodes. The dashed arcs indicate that $v_2 = s(v_3)$, $v_3 = s(v_4)$, and $v_4 = s(v_5)$.

5.3 Modifications to FERP

FERP uses the $f(u,i)$ and $h(u,i)$ values to determine $r(t)$ for each edge t. Because the definition of $r(v_i^{(k)})$ is similar to the definition of $r(t)$, FNRP uses a modification of FERP to compute $r(v_i^{(k)})$. The differences between the definitions of $r(t)$ and $r(v_i^{(k)})$ are that $r(v_i^{(k)})$ is defined for a graph with supernodes, not nodes, and that $r(v_i^{(k)})$ must not be incident on any of v's star nodes. We modify the definitions of $f(u,i)$ and $h(u,i)$ to account for these differences.

5.3.1 Preprocessing

Before the first iteration, for each edge e, FNRP precomputes lca(e), the lowest common ancestor in T^N of the endpoints of e, and the values child1(e) and child2(e), defined as follows. As with FERP, the parallel lowest common ancestor algorithm of Schieber and Vishkin [SV 88] can determine these values in $O(\log n)$ time with m processors. Let $v_1 = \text{lca}(e)$. The two children of v_1 that are in $C^N(e)$ are denoted child1(e) and child2(e), respectively, as in Figure 17. These two values are used to determine $s(v_i^{(k)})$; in addition, these values are used during the FERP computation in the kth iteration for $k > 1$.

Finally, for each node u in T^N, FNRP precomputes anc(u,i), the ancestor of u that is 2^i levels above u in the tree. FNRP uses pointer jumping up the tree to determine these values. Hence, with n processors, these values can be determined in $O(\log n)$ time.

After the anc(u,i) values are precomputed, FNRP can determine z, the ancestor of u at

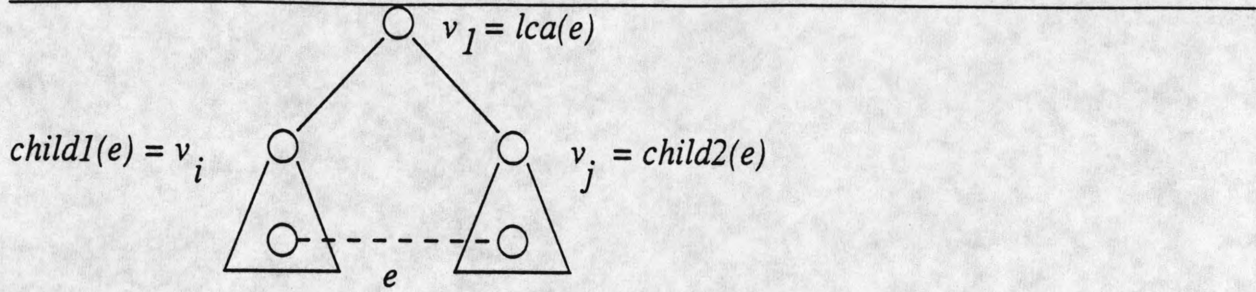

Figure 17: Example of child1(e) and child2(e).

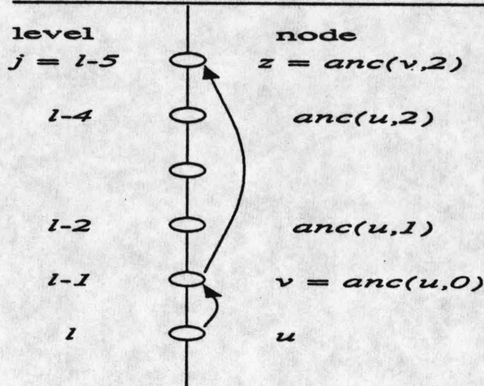

Figure 18: Precomputing and looking up ancestors.

some level j in $O(\log n)$ time using one processor. Let $l = \text{level}(u)$. First, FNRP computes the binary representation of $l-j$, and then FNRP uses the ones in this representation to look up z. For example, suppose $l-j=5$, which is 101_2 in binary. FNRP looks up $v = \text{anc}(u,0)$, and then looks up $z = \text{anc}(v,2)$. Figure 18 illustrates this example.

5.3.2 Modifications During First Iteration

For the first iteration, $G^{(1)} = G^N$ and each "supernode" is just one node. Hence, we need only to modify $f(u,i)$ and $h(u,i)$ to ensure that $r(v_i)$ is not incident on any of v's star nodes. FNRP adds this constraint to the definitions of $f(u,i)$ and $h(u,i)$ as follows:

- Instead of $f(u,i)$, FNRP uses

$$f(u,i,\overline{z}) = \min\{w(e) | e = (u,v), \text{level}(e) \leq i, e \notin E^N(z)\}.$$

- Instead of $h(u,i)$, FNRP uses

$$h(u,i,\overline{z}) = \min\{w(e) | e = (x,y), x \in T_u^N, e \notin E^N(z)\}.$$

- Instead of setting $r(t)$ to be the edge with weight $h(v, \text{level}(v)-1)$, FNRP determines $r(v_i)$ to be the edge with weight

$$h(v_i, \text{level}(v_i)-1, \overline{v}) = h(v_i, \text{level}(v_1), \overline{v}).$$

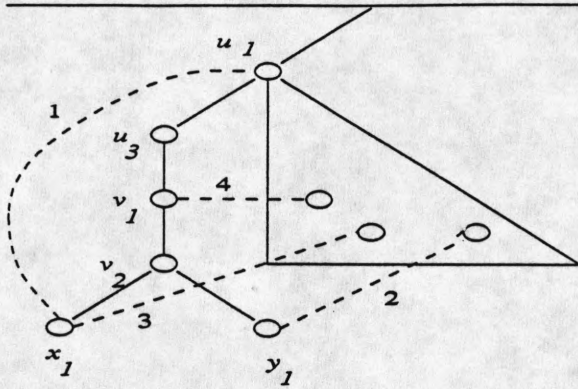

Figure 19: Modified definitions for $f(u, i, \overline{z})$ and $h(u, i, \overline{z})$.

Figure 19 illustrates these definitions. Let level$(u_1) = \ell$. For x_1, $f(x_1, \ell, \overline{u}) = 3$, not 1, because edge 1 is incident on u_1. The value of $h(v_1, \ell, \overline{u})$ depends on the f values in v_1's subtree: $f(v_1, \ell, \overline{u}) = 4$ and $f(y_1, \ell, \overline{u}) = 2$, so $h(v_1, \ell, \overline{u}) = \min\{2, 3, 4\} = 2$. Because v_2 has no nontree edges incident on it, $f(v_2, \ell, \overline{u}) = \texttt{nil}$. Finally, $r(u_3) = h(u_3, \ell, \overline{u}) = h(v_1, \ell, \overline{u}) = 2$, where the second equality follows because u_3 has no incident nontree edges.

As with $f(u, i)$ in Section 4.4 , FNRP precomputes only $\delta_E(u)$ values of $f(u, i, \overline{z})$ for each node u: FNRP determines $f(u, i_l, \overline{z})$ only if u has an incident nontree edge with level i_l. This computation is similar to the precomputation for $f(u, i_l)$. Let $M(u, i_l)$ be the set of nontree edges with level i_l that are incident on u. FNRP determines the minimum weight edge in $M(u, i_l) - \{(u, z)\}$. Next, FNRP uses a parallel prefix algorithm on these values to compute $f(u, i_l, \overline{z})$. In total, for all u, this computation takes $O(\log n)$ time, using m processors on a CREW PRAM.

For a general value of i, FNRP looks up $f(u, i, \overline{z})$ in $O(1)$ time using $\delta_E(u)$ processors on a CREW PRAM, also as in Section 4.4. Each processor is assigned to one level. For example, suppose u has nontree edges incident with levels i_1 and i_2, $i_1 < i_2$, and suppose that $i_1 \leq i < i_2$. The processor assigned to i_1 looks up i_1 and i_2, notes that $i_1 \leq i < i_2$, and returns $f(u, i, \overline{z})$.

Once the values of $f(u, i, \overline{z})$ have been determined, FNRP runs FERP to compute the values of $h(v_i, \text{level}(v_i) - 1, \overline{v})$ in $O(\log n)$ time. Then $r(v_i)$ is the edge with weight $h(v_i, \text{level}(v_i) - 1, \overline{v})$.

Next, given v_i and $r(v_i)$, FNRP determines $s(v_i)$. If $v_1 \neq \text{lca}(e)$, then $s(v_i) = v_1$. Otherwise, when $v_1 = \text{lca}(e)$, without loss of generality, let $v_i = \text{child1}(e)$; in this case, $s(v_i) = \text{child2}(e)$. Hence, FNRP makes one lca query and at most two child queries for v_i, for a total of $O(\delta_T(v))$ queries for v's star. Using n processors for all nodes, one processor per node, these queries take $O(1)$ time.

To merge nodes, FNRP uses the $s(v_i)$ values. Since $\sum \delta_T(v) = 2(n-1)$, with pointer jumping, FNRP can merge nodes for all stars simultaneously in $O(\log n)$ time with n processors. FNRP just assigns a common representative to each node in a chain, as defined by the $s(v_i)$ values.

In total, the first iteration takes $O(\log n)$ time.

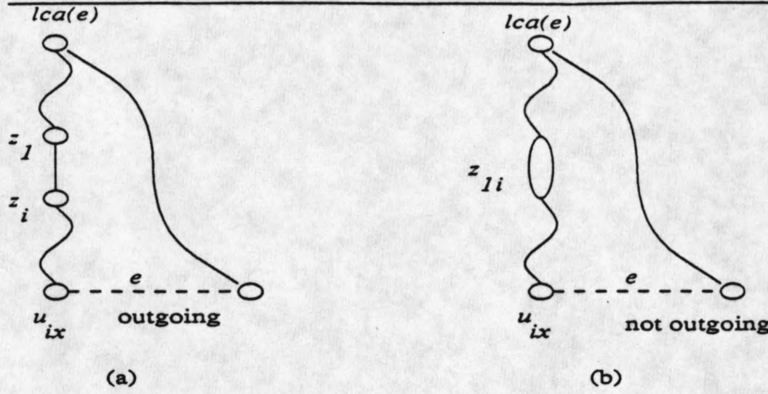

Figure 20: Two possibilities for $z_1 \neq \text{lca}(e)$: (a) z_1 and z_i not in same supernode; (b) z_1 and z_i in supernode z_{1i}.

5.3.3 Modifications During kth Iteration, $k > 1$

For the kth iteration, $k > 1$, FNRP has to modify $f(u, i, \overline{z})$ and $h(u, i, \overline{z})$ to handle supernodes in $G^{(k)}$. FNRP first determines $f^{(k)}(u_i^{(k)}, j, \overline{z})$ from values $f^{(k)}(u_{ix}, j, \overline{z})$ computed for each node u_{ix} contained in supernode $u_i^{(k)}$; by doing so, we avoid having to update the adjacency lists for each graph $G^{(k)}$. Then FNRP runs FERP on $G^{(k)}$ to compute the $h^{(k)}(u_i^{(k)}, j, \overline{z})$ values.

In this iteration, FNRP wants the minimum outgoing edge for each component in $CG^{(k)}(z)$. Edge e is *outgoing* with respect to z if e is not a self-loop in $CG^{(k)}(z)$. FNRP has to check whether each edge considered for $f^{(k)}(u_{ix}, j, \overline{z})$ is outgoing with respect to z. After checking every nontree edge incident on u, FNRP computes $f^{(k)}(u_{ix}, j, \overline{z})$ for each node u_{ix} in $u_i^{(k)}$, where

$$f^{(k)}(u_{ix}, j, \overline{z}) = \min\{w(e) | e = (u_{ix}, v), \text{level}(e) \leq i, e \notin E^N(z), e \text{ outgoing w.r.t. } z\}.$$

Next, FNRP takes the minimum of these values for $f^{(k)}(u_i^{(k)}, j, \overline{z})$:

$$f^{(k)}(u_i^{(k)}, j, \overline{z}) = \min\{f^{(k)}(u_{ix}, j, \overline{z}) | u_{ix} \in u_i^{(k)}\}.$$

Other than checking each edge e considered for $f^{(k)}(u_{ix}, j, \overline{z})$, the computation is similar to the ones for $f(u, i)$ and $f(u, i, \overline{z})$.

To determine whether edge e is outgoing in $CG^{(k)}(z)$, FNRP considers two cases: either $z_1 \neq \text{lca}(e)$ or $z_1 = \text{lca}(e)$, where z_1 is the central node of z's star. In the first case, $z_1 \neq \text{lca}(e)$, $C^N(e)$ contains only two star nodes of z, namely, z_1 and z_i for some i, as shown in Figure 20(a). If z_1 and z_i have not been merged before iteration k, as in Figure 20(a), then e is an outgoing edge in $CG^{(k)}(z)$. Otherwise, as indicated in Figure 20(b) by the supernode z_{1i}, e is not an outgoing edge. In terms of the notation for FNRP, e is an outgoing edge if and only if $\text{rep}^{(k)}(z_1) \neq \text{rep}^{(k)}(z_i)$. To check this constraint, FNRP has to determine which node $z_i, i > 1$, is an ancestor of u_{ix}. Though FNRP does not know z_i, FNRP knows $\text{level}(z_i) = \text{level}(z_1) + 1$. Consequently, FNRP looks up the ancestor of u_{ix} at $\text{level}(z_1) + 1$ using the anc information and then checks $\text{rep}^{(k)}(z_1)$ and $\text{rep}^{(k)}(z_i)$.

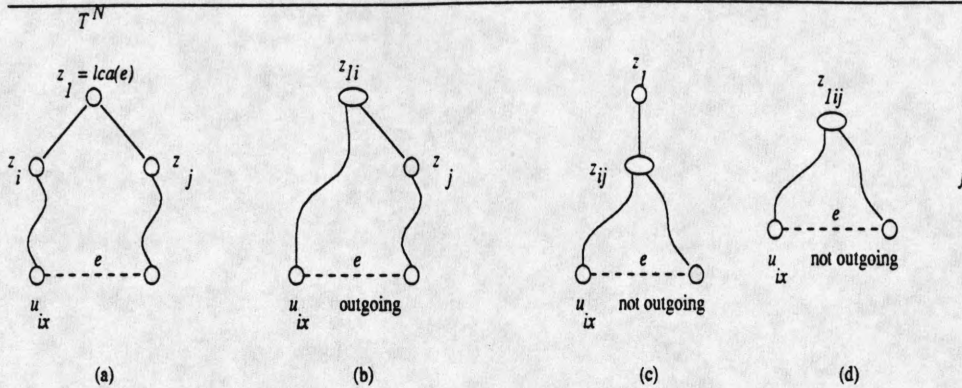

Figure 21: Original cycle and three possibilities for $z_1 = \text{lca}(e)$.

In the second case, $z_1 = \text{lca}(e)$, $C^N(e)$ contains three star nodes of z, namely, z_1, z_i, and z_j, as shown in Figure 21(a). If z_1 and z_i, but not z_j, have not been merged, as in Figure 21(b), then e is an outgoing edge. Similarly, but not shown, if z_1 and z_j, but not z_i, have been merged, then e is an outgoing edge. On the other hand, if z_i and z_j, but not z_1, have been merged (Figure 21(c)), or if all three have been merged (Figure 21(d)), then e is not an outgoing edge. In terms of FNRP's notation, e is an outgoing edge if and only if $\text{rep}^{(k)}(z_i) \neq \text{rep}^{(k)}(z_j)$. To check this constraint, FNRP looks up $z_i = \text{child1}(e)$ and $z_j = \text{child2}(e)$ and then looks up $\text{rep}^{(k)}(z_i)$ and $\text{rep}^{(k)}(z_j)$.

For each endpoint of an edge e, FNRP makes one lca query and either $O(\log n)$ anc queries or two child queries. Hence, using m processors, FNRP takes $O(\log n)$ time to check whether edges are outgoing.

Once the $f^{(k)}(u_i^{(k)}, j, \overline{z})$ values are determined, FNRP computes

$$h^{(k)}(u_i^{(k)}, j, \overline{z}) = \min\{w(e) | e = (x, y), x \in T_u^{(k)}, e \notin E^N(z)\}$$

in $O(\log n)$ time. The rest of the kth iteration is the same as the first iteration, with a slight modification to determine $s(v_i^{(k)})$. Let $e = r(v_i^{(k)})$. If $v_1 \neq \text{lca}(e)$, then $s(v_i^{(k)}) = v_1^{(k)}$. Otherwise, FNRP looks up the values child1(e) and child2(e). Without loss of generality, let child1(e) be in $v_i^{(k)}$; consequently, $s(v_i^{(k)})$ is the supernode that has $\text{rep}^{(k)}(\text{child2}(e))$ as its representative. As in the first iteration, determining $s(v_i^{(k)})$ takes $O(1)$ time.

In total, the kth iteration takes $O(\log n)$ time using m processors.

5.4 Correctness

We first prove the FNRP invariant, which formalizes the relationship between $G^{(k)}$ and each component graph.

Lemma 3 (FNRP Invariant) *For each k and node v in V, the star nodes $v_{i1}, \ldots, v_{il}$ comprise one supernode in $G^{(k)}$ if and only if they comprise one supernode in $CG^{(k)}(v)$,*

Proof: We prove the lemma by induction on k. The base case, $k = 1$, follows from the correspondence between $CG(v)$ and $G^{(1)} = G^N$.

For the inductive step, we assume the hypothesis holds for k and show that it holds for $k+1$. In $G^{(k)}$, FNRP merges supernodes that are connected by the $s(v_i^{(k)})$ relationship. Let $v_i^{(k+1)}$ be a supernode formed by the merging of supernodes in $G^{(k)}$, and suppose that $v_{i1}, \ldots, v_{il}$ are the star nodes that it contains. By the inductive hypothesis and the definition of $s(v_i^{(k)})$, the supernodes in $CG^{(k)}$ that contain $v_{i1}, \ldots, v_{il}$ comprise one connected component. Hence, these supernodes are merged into one supernode in $CG^{(k+1)}$. □

With the modifications to the definitions of $f(u, i)$ and $h(u, i)$, FNRP ensures that $r(v_i^{(k)})$ is not incident on any of v's star nodes and that $r(v_i^{(k)})$ is an outgoing edge from $v_i^{(k)}$. The correctness of FERP implies that $r(v_i^{(k)})$ is the minimum weight nontree edge that contains the edge $(v_i^{(k)}, p(v_i^{(k)}))$ in its cycle. The last part of the correctness proof of FNRP is to show that $r(v_i^{(k)})$ corresponds to the minimum weight outgoing edge from $v_i^{(k)}$ in $CG^{(k)}(v)$.

Lemma 4 *For each iteration k, for each supernode $v_i^{(k)}$, $r(v_i^{(k)})$ corresponds to the minimum weight outgoing edge from $v_i^{(k)}$ in $CG^{(k)}(v)$.*

Proof: Let e be the minimum weight outgoing edge from $v_i^{(k)}$ in $CG^{(k)}(v)$. The crux of the proof is showing that the edge $(v_i^{(k)}, p(v_i^{(k)}))$ is in $C^{(k)}(e)$. For $i > 1$, the claim is simple to prove. In this case, because e is an outgoing edge for $v_i^{(k)}$, $C^{(k)}(e)$ contains star edges from $v_i^{(k)}$ to the other endpoint of e in $CG^{(k)}(v)$. Every tree path from $v_i^{(k)}$ to another of v's star supernodes passes through $(v_i^{(k)}, v_1^{(k)})$.

For $i = 1$, let $x = p(v_1^{(k)})$ in $T^{(k)}$. The unique tree path from x to another of v's star supernodes must pass through $v_1^{(k)}$, hence $(v_1^{(k)}, x)$ is in $C^{(k)}(e)$.

Finally, FERP ensures that $r(v_i^{(k)})$ is the minimum weight edge such that its cycle contains $(v_i^{(k)}, p(v_i^{(k)}))$. That is, $r(v_i^{(k)})$ and the minimum outgoing weight edge from $v_i^{(k)}$ in $CG^{(k)}(v)$ are the same edge. □

As a corollary of these two lemmas, FNRP determines the correct node replacement for each node.

Theorem 2 *FNRP determines the correct node replacement for each node in V.*

6 Comments

Throughout the paper, we assume that G is biconnected. If G is not biconnected, then the sequential algorithms terminate when $k = m - n + 1$. In this case, replacements are determined for the biconnected components of G. In each stage k, T_k is a may be a forest of trees instead of one single tree, and the final "tree" T_{m-n+2} consists of the bridge edges and distinct components of G. For example, suppose that edge e is a bridge in G; after termination of, say, FER, the final tree T_{m-n+2} consists of one edge that corresponds to e.

Another problem related to AER and ANR is All Edge Additions (AEA): determine the MST of $G+(u, v)$, for each edge (u, v) not in E, simultaneously. The algorithm for AEA runs recursively on the tree edges. First sort the tree edges by weight; this step takes $O(n \log n)$

time since there are $n-1$ tree edges. Let $t^*(x,y)$ be the heaviest tree edge. Consider the two components of T produced when t^* is deleted; call these components T_x and T_y, respectively. For each node pair u, v, with u in T_x and v in T_y, there are two possibilities for the MST of $G+(u,v)$. If $w(u,v) \geq w(t^*)$, then T is the MST of $G+(u,v)$. Otherwise, if $w(u,v) < w(t^*)$, then the MST of $G+(u,v)$ is $T - t^* + (u,v)$. The algorithm then calls itself on the two components T_x and T_y.

The algorithm for AEA produces the following output: for each node pair u, v such that edge (u,v) is not in E, the algorithm determines the tree edge t^* that would be displaced from the MST by adding (u,v) with $w(u,v) < w(t^*)$. Hence, the total running time is proportional to the number of node pairs, $O(n^2)$.

7 Open Questions

- For the input to the parallel algorithms, the edges do not have to be sorted by weight. The algorithm of Dixon *et al.* is an $O(m)$ time algorithm for AER that does not require edges to be sorted first. Are there $O(m)$-time sequential algorithms for ANR that do not require the edges to be sorted first?

- Are there analogous results for other basic graph problems? For example, what is the total time needed to compute the shortest paths between all pairs of nodes, in each graph $G-v$, for all v simultaneously? There is a naive sequential algorithm that runs in time $O(m^*n^2+n^3\log n) = n \times SP$, where SP is the time complexity of the sequential all-pairs shortest path algorithms of Karger *et al.* [KKP 93] and McGeoch [Mc 95].

Acknowledgements

This research was supported in part by the National Science Foundation under Grant CCR-9315696. We thank D. Eppstein and R. Tarjan for pointing out the paper of Dixon *et al.*, N. Amato for suggesting that we design parallel algorithms for AER and ANR, D. Brown for providing insightful comments, K. Walny for highlighting refinements to the implementation of FER, and S. Pemmaraju for suggesting the all edge additions problem.

References

[BGH 65] A. Berge and A. Ghouila-Houri, *Programming, Games, and Transportation Networks*, John Wiley, New York, 1965.

[CH 78] F. Chin and D. Houck, "Algorithms for updating minimal spanning trees," *J. Comput. System Sci.*, **16**, 3 (1978) 333–344.

[Co 88] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, **17**, 4 (Aug 1988) 770–785.

[CLR 92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1992.

[DF 94] S. K. Das and P. Ferragina, "An $o(n)$ Work EREW Parallel Algorithm for Updating MST," In *Proceedings of the Second Annual European Symposium on Algorithms*, Utrecht, The Netherlands, Sept 1994, 331–342.

[DRT 92] B. Dixon, M. Rauch, and R. E. Tarjan, "Verification and sensitivity analysis of minimum spanning trees in linear time," *SIAM J. Comput.*, **21**, 6 (Dec 1992) 1184–1192.

[DT 94] B. Dixon and R. E. Tarjan, "Optimal Parallel Verification of Minimum Spanning Trees in Logarithmic Time," In *Proceedings of the First Canada-France Conference on Parallel and Distributed Computing*, Montreal, Canada, May 1994, 13–22.

[EGIN 92] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification—a technique for speeding up dynamic graph algorithms," in *Proceedings of the 33rd Symposium on Foundations of Computer Science*, (1992) 60–69.

[Fr 85] G. N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications," *SIAM J. Comput.*, **14**, 4 (Nov 1985) 781–798.

[FT 87] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, **34**, 3 (July 1987) 209–221.

[GT 85] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *J. Comput. System Sci.*, **30**, 2 (1985) 209–221.

[HT 84] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.*, **13**, 2 (May 1984) 338–355.

[Ja 92] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[KR 95] S. Kapoor and H. Ramesh, "Algorithms for enumerating all spanning trees of undirected and weighted graphs," *SIAM J. Comput.*, **24**, 2 (Apr 1995) 247–265.

[KKP 93] D. R. Karger, D. Koller, and S. J. Phillips, "Finding the hidden path: time bounds for all-pairs shortest paths," *SIAM J. Comput*, **22**, 6 (Dec 1993) 199-1217.

[Kr 56] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," In *Proceedings of the American Mathematical Society*, **7** (1956) 48–50.

[MSV 86] Y. Maon, B. Schieber, and U. Vishkin, "Parallel ear decomposition search (EDS) and *st*-numbering in graphs," *Theoret. Comput. Sci.*, **47**, 3 (1986) 277–298.

[Mc 95] C. C. McGeoch, "All-pairs shortest paths and the essential subgraph," *Algorithmica*, **13**, 5 (1995) 426–441.

[PL 94] R. Pasquini and M. C. Loui, "A fault tolerant distributed algorithm for minimum-weight spanning trees," Tech. Rep. UILU-ENG-94-2210, Coordinated Sci. Lab., Univ. of Illinois at Urbana-Champaign, Urbana, Ill., 1994.

[PK 93] S. Pawagi and O. Kaser, "Optimum parallel algorithms for multiple updates of minimum spanning trees," *Algorithmica*, **9**, 4 (1993) 357–381.

[Pr 57] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, **36** (1957) 1389–1401.

[RHK 95] M. Rauch Henzinger and V. King, "Randomized dynamic graph algorithms with polylogarithmic time per operation," in *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, (1995) 519–527.

[RMM 93] M. Reid-Miller, G. L. Miller, and F. Modugno, "List Ranking and Parallel Tree Contraction," Chapter 3 in *Synthesis of Parallel Algorithms* (ed. J. Reif), Morgan Kaufmann, 1993.

[SV 88] B. Schieber and U. Vishkin, "On finding lowest common ancestors: simplification and parallelization," *SIAM J. Comput.*, **17**, 6 (Dec 1988) 1253–1262.

[SP 75] P. M. Spira and A. Pan, "On finding and updating spanning trees and shortest paths," *SIAM J. Comput*, **4**, 3 (Sept 1975) 375–380.

[Ta 79] R. E. Tarjan, "Applications of path compression on balanced trees," *J. ACM*, **26**, 4 (Oct 1979) 690–715.