

# Efficient Self-stabilizing Algorithms for Tree Networks

Jean R. S. Blair\*      Fredrik Manne†

## Abstract

Many proposed self-stabilizing algorithms require an exponential number of moves before stabilizing on a global solution, including some rooting algorithms for tree networks [1, 2, 3]. These results are vastly improved upon in [6] with tree rooting algorithms that require only  $O(n^3 + n^2 \cdot c_h)$  moves, where  $n$  is the number of nodes in the network and  $c_h$  is the highest initial value of a variable. In the current paper, we describe a new set of tree rooting algorithms that brings the complexity down to  $O(n^2)$  moves. This not only reduces the first term by an order of magnitude, but also reduces the second term by an unbounded factor. We further show a generic mapping that can be used to instantiate an efficient self-stabilizing tree algorithm from any traditional sequential tree algorithm that makes a single bottom-up pass through a rooted tree. The new generic mapping improves on the complexity of the technique presented in [8].

**Keywords:** Self-stabilizing, tree, root, median, center, bottom-up.

## 1 Introduction

A distributed system can be modeled as an undirected graph  $G = (V, E)$ , where  $V$  is the set of  $n$  systems, or nodes, and  $E$  is the set of links, or

---

\*Department of Electrical Engineering and Computer Science, United States Military Academy, West Point, NY, 10996, USA, [Jean-Blair@usma.edu](mailto:Jean-Blair@usma.edu)

†Department of Informatics, University of Bergen, N-5020 Bergen, Norway, [Fredrik.Manne@ii.uib.no](mailto:Fredrik.Manne@ii.uib.no)

edges, interconnecting the systems together. In the self-stabilizing algorithmic paradigm, each node can only see its neighbors and itself, yet the system of simultaneously running algorithms must converge to a global state satisfying some desired property. In the next section we describe more completely the self-stabilizing algorithmic paradigm.

Numerous results in the literature address the issue of self-stabilizing algorithms for rooting a tree. In [1], [2] and [3] the authors describe tree algorithms for leader election, center-finding and median-finding, respectively. All three algorithms appear to require an exponential number of moves in the worst case. These results are vastly improved upon by the center-finding and median-finding algorithms in [6], both of which require only  $O(n^3 + n^2 \cdot c_h)$  moves, where  $c_h$  is the highest initial value of the variables used in the computation. We improve further on these results by describing a generic self-stabilizing tree rooting algorithm that requires only  $O(n^2)$  moves. This new generic algorithm can be used to solve, among others, the problems of electing a leader in a tree, finding the center of a tree and finding the median of a tree. Section 3 defines these and other graph properties, gives the generic tree rooting algorithm and shows how to choose a leader based on various criteria.

There are numerous graph problems that can be solved on a tree once the tree is rooted. Many self-stabilizing algorithms in the literature rely on this, augmenting an arbitrary tree rooting algorithm to solve one or another prob-

lem. See, for example, [4, 10, 12, 13]. In a seminal work, the authors of [8]<sup>1</sup> present a unified approach to translating linear-time bottom-up sequential dynamic programming tree algorithms into self-stabilizing tree algorithms. Their technique can easily be applied to improve the results of [4, 12, 13]. The analysis in [8] uses round complexity, showing that the resulting tree algorithms require  $2r + 3$  rounds, for a tree of radius  $r$ . It is easy to see that a single round could require as much as  $O(n^2 + n \cdot c_h)$  moves, where  $c_h$  is again the highest initial value. Thus, they have described a set of self-stabilizing tree algorithms that run in  $O(n^3 + n^2 \cdot c_h)$  moves. In Section 4 we present an improved generic mapping from sequential bottom-up one-pass tree algorithms to self-stabilizing tree algorithms that require only  $O(n^2)$  moves. Included in that section is an example of instantiating a self-stabilizing algorithm, in this case to find a maximum independent set in a tree.

The fundamental idea used for all of the results in this paper is the time-honored concept of using a little additional storage in order to drastically reduce the computation time. Where previous algorithms would have used  $n$  variables, one per node in the network, we use  $2n - 2$  variables, two per edge in the network. Our algorithms have the added benefit of greatly simplified proofs of correctness. Section 5 contains a brief discussion of the fundamental difference between our self-stabilizing algorithmic approach and that of the previously published tree algorithms.

---

<sup>1</sup>Also to appear as [11].

## 2 The Self-Stabilizing Paradigm

A distributed system is modeled as undirected graph  $G = (V, E)$ , where  $V$  is a set of  $n$  nodes and  $E$  is a set of  $m$  edges. If  $i$  is a node, then  $N(i)$ , its *open neighborhood*, denotes the set of nodes to which  $i$  is adjacent. Every node  $j \in N(i)$  is called a *neighbor* of node  $i$ . The set  $N[i] = N(i) \cup \{i\}$  is the *closed neighborhood* of  $i$ . The *induced subgraph*  $G - i$  is the subgraph of  $G$  induced by the nodes in  $V - \{i\}$ .

In a *self-stabilizing algorithm*, each node can change the value of its local variables based only on the value of the local variables in its closed neighborhood. The contents of a node's local variables constitute its *local state*. The system's *global state* is the union of all local states. No assumptions can be made about the initial global state; that is, the initial value of every local variable is arbitrary. A node  $i$  changes its local state by making a *move*, i.e., changing the value of one or more of its local variables.

Self-stabilizing algorithms are often given as a set of rules of the form “**if**  $p(i)$  **then**  $m(i)$ ,” where  $p(i)$  is a predicate and  $m(i)$  is a move. The predicate  $p(i)$  is defined in terms of the local state of  $i$  and the local states of its neighbors  $j \in N(i)$ . Every node in the system  $G$  runs the same set of rules. A node  $i$  is *privileged* if at least one of its rule-predicates is true. When a node is privileged, it may execute one of its privileged moves. No assumptions can be made about which privileged move is made when more than one exists. We say the system has *stabilized* if no nodes are privileged.

Analysis of a self-stabilizing algorithm is slightly different than that of a sequential algorithm. For self-stabilizing algorithms it is not the overall time-complexity that is of concern, but rather the number of moves made by the algorithm. That is, the time required to perform checks for predicates or to make updates to local variable values does not matter. The self-stabilizing algorithm is evaluated based only on the number of rules that are fired, or equivalently the number of moves made.

As is common for self-stabilizing algorithms, we will assume that each node has a unique identifier. We will discuss in the conclusion how our algorithms can be modified to not require unique identifiers.

Many self-stabilizing algorithms work correctly only in the presence of a central daemon that serializes the moves made by privileged nodes [7]. Our algorithms do not rely on this restriction. We only assume read-write atomicity. Thus, two or more nodes can make simultaneous moves, as long as no node is accessing a value that is being written by another node.

### **3 A Generic Leader Election Algorithm**

An important problem in distributed computing is that of electing the leader in a graph. When the graph is a tree this is equivalent to designating the root of a tree. We will assume throughout the remainder of this paper that the graph  $G$  is a tree. In this section we describe a generic self-stabilizing leader election algorithm for trees.

The generic algorithm can be instantiated with any rooting property that is based on a distance-from-the-leaves property of the nodes in the tree, as long as a tree can have only one “winner” (or two “winners”, if the two can each detect the presence of a second) with the best distance-from-the-leaves. The following is a partial list of such properties.

- Maximum distance to a leaf. The *center* of a tree is a node whose maximum distance to a leaf is minimum. A tree has either a unique center or two adjacent centers[9].
- Sum of distances to all other nodes. The *median* of a tree is a node for which the sum of the distances from it to all other nodes is minimum. A tree has either a unique median or two adjacent medians[9].
- Size of largest disconnected component after removal. An  $\frac{n}{2}$ -*separator* of a tree is a node whose removal results in two or more disconnected components, each of which has no more than  $\frac{n}{2}$  nodes in it. A tree has either a unique  $\frac{n}{2}$ -separator or two adjacent  $\frac{n}{2}$ -separators[9].

The generic leader election algorithm can be viewed as consisting of two separate phases. In the first phase each node  $i$  determines, for each  $j \in N(i)$ , the particular distance property for the node  $i$  in the component of  $G - j$  that contains  $i$ . In the second phase each node determines if it is the one node (or if it and a neighbor of its are the two nodes) in the tree with the desired property. The “output” of the algorithm is that exactly one node

will have designated itself as the leader, and all other nodes will know which neighbor of theirs is closer to the leader than they are.

In the following two subsections we describe the two phases of a self-stabilizing algorithm that finds an  $\frac{n}{2}$ -separator of a tree. Note that for a tree, a  $\frac{n}{2}$ -separator is also a median. Thus, as presented the algorithm solves the median-finding problem. In the third subsection we discuss how to modify the rules presented here for a generic self-stabilizing rooting tree algorithm and how to instantiate a center-finding tree algorithm.

### 3.1 Phase one – determining distances from leaves

For this presentation, the distance-from-the-leaves property is the size of the largest subtree. To that end, each node  $i$  maintains an array  $size_i$  of integer variables. The array contains one value  $size_i(j)$  for each  $j \in N(i)$ . Once the system stabilizes,  $size_i(j)$  will contain the number of nodes in the connected component of  $G - j$  containing  $i$ . Note that only the local variable  $size_i(j)$  in node  $i$  will be accessed by node  $j \in N(i)$ , and thus, as is assumed in the self-stabilizing paradigm, no node needs to have knowledge of the existence of nodes that are a distance of two or more from it.

The algorithm described by the rule R1 below runs on each node of the network. We will later show that once the network has stabilized, each node  $i$  can determine the number of nodes in the entire network by computing  $size_i(j) + size_j(i)$  for any neighbor  $j \in N(i)$ . Each node interprets the rule by substituting itself in place of  $i$ .

**R1:**   **if**    $(\exists j \in N(i) \text{ such that } size_i(j) \neq 1 + \sum_{k \in N(i) - \{j\}} size_k(i))$   
           **then**    $size_i(j) \leftarrow 1 + \sum_{k \in N(i) - \{j\}} size_k(i)$

Note that if  $i$  is a leaf, then  $size_i(j)$  will be set to 1 by R1.

The proof of correctness of this algorithm uses several straight-forward inductive arguments to show that the number of times that the value of  $size_i(j)$  changes is no more than the number of nodes in the component of  $G - j$  that contains  $i$ . Due to space limitations, the four lemmas leading to the following theorem and their proofs of correctness are omitted here. The interested reader is referred to [5] for these and other proofs omitted from this paper.

**Theorem 3.1** *For any given tree network, the distributed algorithm R1 stabilizes in no more than  $n(n - 1)$  moves with  $size_i(j)$  equal to the number of nodes in the component of  $G - j$  that contains  $i$  for every pair of adjacent nodes  $i$  and  $j$ .*

**Lemma 3.2** *The bound of  $n(n - 1)$  moves for R1 is tight.*

### 3.2 Phase two – rooting the tree

The algorithm for rooting the tree is based on each node having determined its position relative to a  $\frac{n}{2}$ -separator in  $G$ . To accomplish this, rule R1 (defined in the previous subsection) is used so that “first” every node will know the size of the entire tree. A node  $i$  will make the “subsequent” moves of determining its position relative to an  $\frac{n}{2}$ -separator of the tree only when, from its local perspective, it believes it knows the correct size of the tree. The following predicate is evaluated for this purpose:



$$sizeCorrect_i = (\forall j \in N(i), size_i(j) = 1 + \sum_{k \in N(i) - \{j\}} size_k(i))$$

Note that when  $sizeCorrect_i$  is true rule R1 cannot be applied at node  $i$ . The following result shows how a node  $i$  can determine the number of nodes in the graph once  $sizeCorrect_i$  evaluates to true.

**Lemma 3.3** *If  $sizeCorrect_i$  is true then  $size_i(j) + size_j(i) = size_i(k) + size_k(i) = n$  for all pairs of nodes  $j, k \in N(i)$ .*

Thus when node  $i$  stabilizes locally it can calculate the number of nodes it assumes that the graph contains as  $n_i = size_i(j) + size_j(i)$  for any  $j \in N(i)$ . Note that Lemma 3.3 implicitly proves that when R1 has globally stabilized  $n_i = n$  for each node  $i$ . In this case it is straightforward for node  $i$  to determine if it is an  $\frac{n}{2}$ -separator and if not to determine its position relative to the separator by locating its neighbor  $j$  with the largest  $size_j(i)$  value.

In addition to the array  $size_i$  of integer variables, each node also maintains a parent pointer  $p_i$ . When the algorithm terminates, the value of  $p_r$  for the root node  $r$  will be  $r$ , while the value of  $p_i$  for each node  $i \neq r$  will be the neighbor of  $i$  that is closest to  $r$ .

The algorithm is given by rule R1 together with rules R2–R5 below. The R2–R5 predicates all require that  $sizeCorrect_i$  be true; thus they can be applied only if R1 cannot be applied. The purpose of these rules are as follows. R2 is used to identify the  $\frac{n}{2}$ -separator if a unique one exists, and to make the unique separator be the root. R3 is used by nodes that are not  $\frac{n}{2}$ -separators to set their pointer to their closest-to-the-separator neighbor. R4 and R5 are used in the case where there are two  $\frac{n}{2}$ -separators. R4 selects

as the root the  $\frac{n}{2}$ -separator with higher unique ID, while R5 sets the pointer of the  $\frac{n}{2}$ -separator with lower unique ID to point to its  $\frac{n}{2}$ -separator neighbor. The value  $size_i(j) + size_j(i)$  for an arbitrary  $j \in N(i)$  is used for  $n_i$ .

- R2:**   **if**   ( $sizeCorrect_i$ )  
              **and** ( $\forall j \in N(i) \ size_j(i) < n_i/2$ )  
              **and** ( $p_i \neq i$ )  
              **then**   ( $p_i \leftarrow i$ )
- R3:**   **if**   ( $sizeCorrect_i$ )  
              **and** ( $\exists j \in N(i)$  such that  $size_j(i) > n_i/2$ )  
              **and** ( $p_i \neq j$ )  
              **then**   ( $p_i \leftarrow j$ )
- R4:**   **if**   ( $sizeCorrect_i$ )  
              **and** ( $\exists j \in N(i)$  such that  $size_j(i) = n_i/2$ )  
              **and** ( $ID_i > ID_j$ )  
              **and** ( $p_i \neq i$ )  
              **then**   ( $p_i \leftarrow i$ )
- R5:**   **if**   ( $sizeCorrect_i$ )  
              **and** ( $\exists j \in N(i)$  such that  $size_j(i) = n_i/2$ )  
              **and** ( $ID_i < ID_j$ )  
              **and** ( $p_i \neq j$ )  
              **then**   ( $p_i \leftarrow j$ )

Again, the reader is referred to [5] for statement of the lemmas and proofs that lead to the following result.

**Theorem 3.4** *For any given tree network, the distributed algorithm R1–R5 stabilizes in no more than  $2n^2 - n$  moves with the values of  $p_i$  for all nodes  $i$  defining a rooted tree with an  $\frac{n}{2}$ -separator as the root.*

### 3.3 Other criteria for leaders

In this subsection we outline a generic version of our rooting algorithm and then briefly discuss how the  $\frac{n}{2}$ -separator algorithm can be modified to find the center of a tree.

In the generic version of the rooting algorithm we maintain, at each node  $i$ , an array  $dist_i$  rather than the array  $size_i$ . The value  $dist_i(j)$  for each  $j \in N(i)$  will contain the value of the distance-from-the-leaves property in the connected component of  $G-j$  containing  $i$ , once the system has stabilized. Let the function  $dfl(\cup_{k \in N(i)-\{j\}} dist_k(i))$  be one that calculates the value of the distance-from-the-leaves property for a node  $i$  based on the value of the distance-from-the-leaves property of all “children” neighbors (i.e., all neighbors  $k \neq j$ ). Then rule R1 can be re-written as follows.

**R1:**   **if**    $(\exists j \in N(i) \text{ such that } dist_i(j) \neq dfl(\cup_{k \in N(i)-\{j\}} dist_k(i)))$   
          **then**    $dist_i(j) \leftarrow dfl(\cup_{k \in N(i)-\{j\}} dist_k(i))$

It is easy to see that the  $\frac{n}{2}$ -separator algorithm given in Subsection 3.1 is this generic algorithm instantiated with  $dfl(\cup_{k \in N(i)-\{j\}} dist_k(i)) = 1 + \sum_{k \in N(i)-\{j\}} dist_k(i)$ . Furthermore, the algorithm given in [6] for finding the median of a tree makes this same calculation. The difference is that the algorithm in [6] calculates only one  $dfl$  value for the currently assumed parent of node  $i$ , rather than calculating a  $dfl$  value for every possible parent. Since our algorithm stabilizes with each node having full knowledge of which node is its parent, our algorithm clearly calculates also the median of a tree. It is worth highlighting the difference between the two median-finding algorithms:

our algorithm maintains more values than does the algorithm in [6], but does so in only  $O(n^2)$  moves as compared to the  $O(n^3 + n^2 \cdot c_h)$  moves required by the algorithm in [6].

In a similar fashion we can utilize the criteria in [6] that finds the center of a tree in order to instantiate an  $O(n^2)$  self-stabilizing algorithm. Here the parameter function  $dfi(\cup_{k \in N(i) - \{j\}} dist_k(i)) = 1 + \max_{k \in N(i) - \{j\}} \{dist_k(i)\}$  for both leaves and internal nodes<sup>2</sup> results in our rule R1 giving a similar center finding algorithm to that given in [6]. The difference is again that our algorithm calculates a  $dfi$  for every possible parent, but does so in fewer moves in the worst case.

The rooting algorithms in [6] contain only their version of rule R1. They prove that then the node (or two nodes) that is (are) the leader(s), either median or center, can determine its (their) leader-status by comparing its distance-from-the-leaves property with that of all of its (their) neighbors. We have gone further in Subsection 3.2 by explicitly accomplishing this determination of who the leader is as well as setting pointers for the corresponding rooted tree with rules R2 through R5. These rules can be generalized in the same way that we have generalized rule R1 above by defining generic parameter functions for determining whether or not a node is the leader and substituting those generic parameter functions in for the second condition of each of rules R2 through R5. In the interest of space, we omit these generalizations here.

---

<sup>2</sup>This assumes the function  $\max_{k \in N(i) - \{j\}} dist_k(i)$  evaluates to zero when  $N(i) = \{j\}$ .

## 4 A Generic Bottom-Up Algorithm

Many graph problems can be solved efficiently on trees in a bottom-up fashion. The main idea is to first root the tree at some node  $r$ , and then to propagate computation from the leaf nodes towards  $r$  as follows. Let  $child(i)$  be the set of node  $i$ 's children in the rooted tree. We assume that a function  $f(j)$  has been computed for each  $j \in child(i)$ . Then  $f(i) = g(\cup_{j \in child(i)} f(j))$  for some function  $g$ . If  $i$  is a leaf then  $g() = c$  where  $c$  is an appropriate constant. If  $g$  can be computed in  $O(1)$  time then the sequential time of this algorithm is  $O(n)$ . The solution is given by the union of all  $f$  values.

There exists several examples of self-stabilizing algorithms for solving such problems. The broad class of algorithms in [8] solve these problems in  $O(n^3 + n^2 c_h)$  moves, where  $c_h$  is the maximum initialized value of a variable. In [4] a self-stabilizing algorithm is given for finding a maximum matching in a tree. Their algorithm requires  $O(n^2 \cdot h(n))$  moves, where  $h(n)$  moves are needed to root the tree. Thus, combined with our rooting algorithm their algorithm uses  $O(n^4)$  moves. In this section we will show how all of these problems can be solved by a self-stabilizing algorithm in only  $O(n^2)$  time.

The main idea behind our algorithm is to simultaneously solve the problem with each possible node as the root, storing the results for each node  $i$  in an array  $f_i$ . As with the  $size_i$  array, the value of  $f_i(j)$  for  $j \in N(i)$  contains the desired result for node  $i$ , assuming  $j$  is the parent of  $i$  in a rooted tree. Independently and simultaneously we run both the bottom-up

algorithm to calculate the  $f_i$  values and also an instantiated generic rooting algorithm. Combined, the two algorithms will stabilize in  $O(n^2)$  time. When the combined algorithm stabilizes each node knows its relative position to the root and can use its  $f$  values together with the  $f$  values of its neighbors to determine which of its  $f$  values is its solution.

We are now ready to begin describing the new algorithm. Assume that  $G$  is rooted and that  $j$  is the parent of node  $i$ . We first observe that the value of  $f(i)$  is independent of which node is the root of  $G$  as long as it is situated in the component of  $G - i$  that contains  $j$ .

Initially  $G$  is not rooted, but we know that for any edge  $(i, j)$  we must either have  $p_i = j$  or  $p_j = i$ . Thus if for each edge  $(i, j)$  we calculate both  $f(i)$  under the assumption that  $p_i = j$  and similarly  $f(j)$  assuming that  $p_j = i$  we will, once the tree is rooted, have all the  $f$  values needed to compute a global solution. Note that for the root we must set  $f(i) = g(\cup_{k \in N(i)} f_k(i))$ .

We define for each edge  $(i, j) \in E$  variables  $f_i(j)$  and  $f_j(i)$  associated with node  $i$  and  $j$  respectively. We also define a variable  $solution_i$  for each node which contains node  $i$ 's final  $f$  value. Our algorithm consists of rules R1 through R5 of the  $\frac{n}{2}$ -separator algorithm combined with the following three rules.

- R6:**   **if**    $(\exists j \in N(i) \text{ such that } f_i(j) \neq g(\cup_{k \in N(i) - \{j\}} f_k(i)))$   
          **then**    $f_i(j) \leftarrow g(\cup_{k \in N(i) - \{j\}} f_k(i))$
- R7:**   **if**    $(p_i = i)$   
          **and**  $solution_i \neq g(\cup_{j \in N(i)} f_j(i))$   
          **then**    $(solution_i \leftarrow g(\cup_{j \in N(i)} f_j(i)))$

**R8:**   if   ( $p_i \neq i$ )  
           **and** ( $solution_i \neq f_i(p_i)$ )  
           **then** ( $solution_i \leftarrow f_i(p_i)$ )

Note that in rule R6 we assume that a node can detect if it is a leaf and in this case we define  $g() = c$  for some appropriate constant  $c$ .

**Theorem 4.1** *Algorithm R1 – R8 stabilizes after  $O(n^2)$  moves.*

**Proof.** Note first that R6 is a generalization of R1 with the  $dfl$  function replaced by  $g$ . Thus, the number of times an R6 move is made is the same as R1. Rules R7 and R8 may fire once initially and then at most once after each change in  $p_i$  or some neighboring  $f$  value. From the above discussion and Theorem 3.4 it follows that the total number of rule R7 and R8 moves is  $O(n^2)$ . ■

**Theorem 4.2** *Once the distributed algorithm R1 – R8 has stabilized on a tree network, the values of  $solution_i$  solve the associated graph problem.*

**Proof.** The correctness of the algorithm follows from the fact that R6 behaves in the same way as R1. Thus after R1 through R6 have stabilized only one of rules R7 and R8 may fire once for each node to set  $solution_i$  to its correct value. ■

As an example of how this algorithm can be used to solve a particular problem, we consider the problem of finding a maximum independent set in a tree. A sequential algorithm to solve this was given in [14]. The algorithm effectively roots the tree at its center and then processes the nodes

starting from the leaves, working towards the root. Each node is included in the independent set if none of its children are in the set. To convert this to a self-stabilizing algorithm, we simply consider  $f_i$  to be the appropriate boolean value, defining the algorithm with the function set  $g() = \text{TRUE}$  and  $g(\cup_{j \in \text{set}(i)} f_j(i)) = \wedge_{j \in \text{set}(i)} (\neg f_j(i))$  where  $\text{set}(i) = \text{child}(i)$  for non-root nodes and  $\text{set}(i) = N(i)$  for the root. The correctness of this algorithm follows directly from the correctness of the sequential algorithm.

## 5 Concluding Remarks

The main idea behind our more efficient algorithms is to have rules for which updates propagate along *directed* acyclic paths in an undirected tree. In this way one avoids cycles in the updates and thus obtains fast convergence.

All of the previously-published self-stabilizing tree algorithms mentioned in Section 1 have the characteristic that they locally compute a required value at each node based on a current view of which neighbor is the node's parent. Each time the parent changes, as it does frequently up until stabilization of the rooting algorithm, the value is recomputed. It is at this point that information may in fact be lost. If later the local view of the parent is switched back, then the required value must be recomputed even if nothing has changed other than the view of who the parent is. The algorithms presented in this paper avoid this unnecessary re-computation of values by locally maintaining values for each possible parent (e.g.,  $\text{size}_i(j)$ )



for all  $j \in N(i)$ ). Each value is computed in exactly the same way that it is computed in a sequential algorithm to solve the same problem in a directed acyclic fashion. In the self-stabilizing paradigm, however, we compute values based on all possible rooted tree views. Once the actual root is known, each node can easily select its correct required value (e.g., select the  $j$  for which  $size_j(i)$  is maximum).

The use of an array of local variables for each node could cause a problem in the self-stabilizing paradigm because access to the entire array by a neighbor might constitute the neighbor having to know of the existence of nodes that are not its neighbors (i.e., nodes that are a distance two away from it). This is not a problem for our algorithms since using our algorithms each neighbor accesses exactly one of the values in the array (e.g., neighbor  $j$  will access  $size_i(j)$ , but will not access any other values in the  $size_i$  array).

Our algorithms assumed the existence of a static unique identifier  $i$  for each node. Note, however, that in the case where it is not necessary to distinguish between two possible adjacent roots, our algorithms can easily be adapted to work without unique identifiers, as was done in [6]. In this case, we would simply change the predicate for rule R3 in the generic rooting algorithm to check for all possible roots (e.g.,  $size_j(i) \leq n_i/2$ ), rather than a unique root. Rules R4 and R5 would not be needed then.

## References

- [1] G. ANTONOIU AND P. K. SRIMANI, *A self-stabilizing leader election algorithm for tree graphs*, *Journal of Parallel and Distributed Computing*, 34 (1996), pp. 227–232.

- [2] ———, *A self-stabilizing distributed algorithm to find the center of a tree graph*, Parallel Algorithms and Applications, 10 (1997), pp. 237–248.
- [3] ———, *A self-stabilizing distributed algorithm to find the median of a tree graph*, J. Comput. Sys. Sci., 58 (1999), pp. 215–221.
- [4] J. R. S. BLAIR, S. M. HEDETNIEMI, S. T. HEDETNIEMI, AND D. P. JACOBS, *Self-stabilizing maximum matchings*, Congressus Numerantium, 153 (2001), pp. 1521–1529.
- [5] J. R. S. BLAIR AND F. MANNE, *Efficient self-stabilizing algorithms for tree networks*, Tech. Report 232, Department of Informatics, University of Bergen, Norway, 2002. Available at <http://www.iu.uib.no/publikasjoner/texrap/index.shtml>.
- [6] S. C. BRUELL, S. GHOSH, M. H. KARAATA, AND S. V. PEMMARAJU, *Self-stabilizing algorithms for finding centers and medians of trees*, SIAM Journal on Computing, 29 (1999), pp. 600–614.
- [7] E. W. DIJKSTRA, *Self-stabilizing systems in spite of distributed control*, CACM, 17 (1974), pp. 643–644.
- [8] S. GHOSH, A. GUPTA, M. H. KARAATA, AND S. V. PERMMARAJU, *Self-stabilizing dynamic programming algorithms on trees*, in Proceedings of the Second Workshop on Self-Stabilizing Systems, 1995, pp. 11.1–11.15.
- [9] F. HARARY, *Graph Theory*, Addison-Wesley, 1972.
- [10] T. C. HUANG, J. C. LIN, AND H. J. CHEN, *A self-stabilizing algorithm which finds a 2-center of a tree*, Computers and Mathematics with Applications, 40 (2000), pp. 607–624.
- [11] M. H. KARAATA, S. V. PERMMARAJU, S. GHOSH, AND A. GUPTA, *Self-stabilizing algorithms and dynamic programming*, Journal of Parallel and Distributed Computing, (under revision).
- [12] M. H. KARAATA AND K. A. SALEH, *A self-stabilizing algorithm for maximum matching in trees*, in Proceedings of the Joint Conference of Informaiton Sciences, 1996, pp. 113–116.
- [13] ———, *A distributed self-stabilizing algorithm for maximum matching*, Computer Systems Science and Engineering, (2000), pp. 175–180.
- [14] S. MITCHELL, E. COCKAYNE, AND S. HEDETNIEMI, *Linear algorithms on recursive representations of trees*, J. Comput. Systems Sci., 18 (1979), pp. 76–85.