Distributed Maintenance of a Spanning Tree Using Labeled Tree Encoding

Vijay K. Garg and Anurag Agarwal

University of Texas at Austin
Austin, TX 78712-1084

Abstract. Maintaining spanning trees in a distributed fashion is central to many networking applications. In this paper, we propose a self-stabilizing algorithm for maintaining a spanning tree in a distributed fashion for a completely connected topology. Our algorithm requires a node to process $O(1)$ messages of size $O(\log n)$ on average in one cycle as compared to previous algorithms which need to process messages from every neighbor, resulting in $O(n)$ work in a completely connected topology. Our algorithm also stabilizes faster than the previous approaches.

1 Introduction

Fault tolerance is a major concern in distributed systems. The self-stabilization paradigm, introduced by Dijkstra [8], is an elegant and a powerful mechanism for fault tolerance. Self-stabilizing systems tolerate transient *data faults* that can corrupt the state of the system. They ensure that a system starting from any state converges to a legal state provided the faults cease to occur.

Self-stabilizing algorithms for spanning tree construction have been extensively studied. Spanning trees have many uses in computer networks. Once a spanning tree is established in a network, it may be used in broadcast of a message, convergecast, β synchronizer, and many other algorithms. As a result, it is desirable to have an efficient self-stabilizing algorithm for spanning trees. The first algorithm in this area was given in [10, 11] which deals with building BFS tree for a graph. Other algorithms were also proposed for self-stabilizing BFS trees which dealt with different system models and assumptions [1, 2, 4, 15, 16]. Algorithms have also been proposed for other types of trees — such as DFS tree [6] and minimum spanning tree [3]. A survey of the existing self-stabilizing spanning trees can be found in [13].

In this paper, we use an extension of the well-known strategy of *detection* and *reset* [4, 5]. In this strategy, the nodes periodically test if the system is in a legal state and on detection of a fault, carry out the reset strategy. Many self-stabilizing algorithms have *local detection*, i.e., detection by each node corresponds to evaluation of a boolean predicate only on its variables and its neighbors' variables. The reset procedure may be complicated depending upon the application.

Our method is an extension of the above strategy. We view the set of *global* states as the cross-product of the *core* states and the *non-core* states. The core

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 606–616, 2005.

states satisfy the property: There exists a legal state for every *core* state. The *non-core* component of a global state is maintained only for performance reason. Given the *core* component, one could always recreate the *non-core* component. In our algorithm for maintaining a spanning tree, we use Neville's code [18] of the tree as the *core* component and the *parent* structure as the *non-core* component. Given any Neville's code, there exists a unique labeled spanning tree in a completely connected graph. Now assume that our program suffers from a data fault. The data fault could be in the core component or the non-core component. However, every value of the core component results in a valid code. Therefore, in either case, we assume that it is the non-core component that has changed. Upon detecting that the non-core component does not correspond to the core component, we simply reset the non-core component to a value corresponding to the core component. The challenge lies in identifying suitable core and non-core components and efficient detection and reset of the state when information is distributed across the network.

We assume that our system is a completely connected graph on n nodes with ids $1 \dots n$. Such a system could be a network overlaid on a real network. Given proper routing, Internet could also be considered a fully connected topology. In such an overlaid topology, spanning trees can be used for distributing load among participants involved in the computation of a global function. For such applications, the nodes higher in the tree have to perform more computation. As a result, it is important to change the spanning tree over time so that nodes can function at different levels in the tree and every node shares the workload equally in the long run. This requirement rules out maintaining a single tree which is hardcoded in the algorithm. Our algorithm allows the application to maintain any arbitrary tree and facilitates systematically changing of the tree.

Our algorithm is designed for asynchronous message-passing systems, and does not require a central daemon [8] for scheduling decisions. Although some of our assumptions are stronger than the previous work, our algorithm has some significant advantages. In the popular shared memory model [9] for communication used by self-stabilizing spanning tree algorithms, it is assumed that a process can read/write all its shared variables including communication registers. In a completely connected topology, this means that a node can perform operations on $O(n)$ variables in $O(1)$ time which is very unreasonable especially for a message passing system. On the other hand, we assume that every communication step takes one unit of time and in this model, our algorithm stabilizes in $O(d)$ time, where d is an upper bound on the number of times a node appears in the Neville's code. It turns out that d is $O((\log n)/\log\log n)$ with high probability for a randomly chosen code. This leads to a small stabilization time and to our knowledge, it is the best stabilization time achieved by any algorithm in our model.

2 System Model

We assume that the network is a completely connected graph with n *processes* with ids from 1 to n. The processes in the system are referred to as $P_1 \dots P_n$.

```
x[1] = least node with degree 1
for i from 1 to n − 1
      y[i] = parent of x[i]
      delete edge between x[i] and y[i]
      if (degree[y[i]] = 1 ∧ y[i] ≠ n)
            x[i + 1] = y[i]
      else
            x[i + 1] = least node with degree 1
Output y as the Neville's code
```

Fig. 1. Algorithm to compute Neville's code (y) of a labeled tree

```
j = least node with degree 1
for i from 1 to n − 1
      parent[j] = code[i]
      degree[j] − −
      degree[code[i]] − −
      if (degree[code[i]] = 1) then
            j = code[i]
      else
            j = least degree node with degree 1
```

Fig. 2. Algorithm to compute labeled tree from Neville's code

Each *process* maintains some *local variables*. The processes are connected to each other through point to point channels and communicate by passing messages to each other. The channels are assumed to be reliable and asynchronous. The *configuration* c of the system is described by the values of the *local variables* for the processes and the *messages* present in the channels. A *computation step* consists of internal computation and a single communication operation: a send or receive. From now on, we use the term *step* to refer to a computation step. A step a is said to be applicable to a configuration c iff there exists a configuration c' such that c' can be reached from c by a single step a. An execution $E = (c_1, a_1, c_2, a_2, \ldots)$ is an alternating sequence of configurations and steps such that c_i is obtained from c_{i-1} by the execution of the step a_{i-1}.

Our algorithm does not require any assumptions on the message transit time for correctness but for measuring the time complexity of our algorithm, we assume that a message can be received at the destination in the step next to the one in which it was sent. A process executes one step in one unit of time. The stabilization time of the algorithm is then given in terms of the number of time units required by the algorithm to stabilize. The reason for choosing such a model is explained later.

3 Neville's Third Encoding

To maintain a spanning tree, it is sufficient for each process to maintain a pointer to the parent but this method is not self-stabilizing as a fault in one of the parent pointers may result in an invalid structure. In this section, we present a core data structure which can be used to maintain the spanning tree in a self-stabilizing way.

For simplicity we assume that all spanning trees rooted at P_n constitute the set of legal structures. Later we explain how this assumption can be relaxed to allow any node to become the root. We represent a tree through an encoding for labeled trees called the *Neville's third encoding* [7, 18]. In this paper, we refer to Neville's third code simply as Neville's code. Each labeled spanning tree has a one-to-one correspondence with a Neville's code. This code is a sequence of $n-2$ numbers from the set $\{1 \ldots n\}$. For completeness sake, derivation of Neville's code from a labeled spanning tree is discussed. Given a labeled spanning tree with n nodes, the Neville's code can be obtained by deleting $n-1$ edges in the

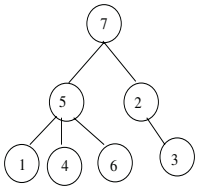

Fig. 3. A spanning tree with Neville's code (5,2,7, 5,5,7)

i	1	2	3	4	5	6	7
$parent$	2	7	5	5	7	5	0
$code$	5	2	7	5	5	7	0
f	2	3	1	4	6	5	7
z	0	2	0	0	5	0	6

Fig. 4. Structures *parent*, *code*, f and z satisfying (R1)-(R5)

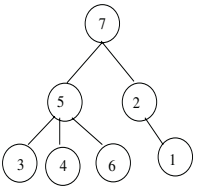

Fig. 5. Tree for *parent* structure given in Fig. 4

tree as shown in Figure 1. The sequence $\{y[i]|1 \leq i \leq n-2\}$ generated at the end of the procedure is called Neville's code.

As an example, consider the labeled tree given in Figure 3. To compute the Neville's code for the tree, we start by deleting the least leaf node, 1. Since the parent of 1 is 5, at this point the code is (5). Now 5 is still not a leaf, so we again choose the least leaf node in the remaining tree, 3. We proceed by deleting 3 and adding its parent 2 to the code. Continuing in a similar fashion, after $n-1=6$ iterations of the algorithm, the code $(5, 2, 7, 5, 5, 7)$ is obtained.

Given Neville's code, the labeled spanning tree can also be computed easily. We first calculate the degree of each node v as one more than the number of times v appears in the code. For the root node n, this gives a value which is one higher than the actual degree of the root but this is intentional. Once the degree of each node is known, the procedure given in Figure 2 can be used to compute the code.

Let Neville's code of the tree be denoted by $code[i]$ for $i \in \{1 \ldots n-2\}$. We require P_i to maintain $code[i]$ as the core data structure and $parent[i]$ as the non-core data structure. If efficiency were not an issue, this would be sufficient for a self-stabilizing algorithm. Periodically, all nodes send their *code* to P_n, P_n calculates $parent[i]$ for each node P_i and sends it back. Then P_i resets $parent[i]$ to the value received from P_n. If $parent[i]$ was corrupted, it gets reset to agree with the spanning tree given by Neville's code. Even if the variable $code[i]$ gets changed, it still results in a valid spanning tree. The *parent* pointers are then reset to agree with the new code.

4 Non-core Data Structures for Spanning Trees

Our strategy is to introduce new data structures in the system so that by imposing a set of constraints on these data structures, we can efficiently detect and correct data faults. For this purpose, the following data structures are used:

- *parent*: The variable $parent[i]$ gives the parent of node P_i in the spanning tree.
- f: The variable $f[i]$ gives us the iteration in which the node P_i is deleted in the Neville's code generation algorithm. Therefore, $code[f[i]]$ gives us $parent[i]$. Since P_n is not deleted in first $n-1$ iterations, we assume that $f[n] = n$.

– z: The variable $z[i]$ gives the largest value of j such that $code[j] = i$. If there is no such j, then $z[i] = 0$.

Based on the properties of Neville's code, it can be verified that the variables — *code*, *parent*, f and z — satisfy the following constraints:

(R1) $\forall i : code[f[i]] = parent[i]$
Follows from the property of function f relating it to the parent.
(R2) $(\forall i : 1 \leq i \leq n-2 \Rightarrow 1 \leq code[i] \leq n) \wedge (code[n-1] = n) \wedge (code[n] = 0)$
Definition of code extended to all the nodes.
(R3) **(1)** $\forall i : 1 \leq i < n \Rightarrow 1 \leq f[i] \leq n-1$
Restricts the f values for nodes other than the root node.
(2) f is a permutation on $[1 \ldots n]$
In each iteration exactly one node is deleted and hence f values are distinct and range from $1 \ldots n$.
(R4) $\forall i : z[i] = \max\{\{j | code[j] = i\} \cup \{0\}\}$
Definition of z.
(R5) $\forall i : z[i] \neq 0 \Rightarrow (f[i] = z[i] + 1)$
If node i was not a leaf node at the starting of the algorithm, then it is deleted immediately after all its children have been deleted.

Theorems 1 and 2 show that constraints are strong enough to characterize a spanning tree, i.e., given a set of data structures *code*, *parent*, f and z which satisfy these constraints, the *parent* structure results in a valid spanning tree regardless of the definitions of these data structures. From now on, when we consider the data structures *code*, *parent*, f and z, we just think of them as obeying a certain set of constraints and not necessarily corresponding to the original definitions that were given for them.

We deal with two sets of constraints — $\mathcal{R} = \{R1, R2, R3(1), R4, R5\}$ and $\mathcal{C} = \{R1, R2, R3, R4, R5\}$. It is evident that any algorithm which satisfies the constraint set $\mathcal{C}$ also satisfies the constraint set $\mathcal{R}$. The trees resulting from obeying these constraint sets possess different guarantees and are characterized by the following theorems.

Theorem 1. *If code, parent, f and z satisfy constraint set $\mathcal{R}$ then parent data structure forms a valid spanning tree rooted at P_n.*

Proof. Let the directed graph formed by the *parent* relation satisfying constraints $\mathcal{R}$ be T_{parent}. The edges of T_{parent} are directed from the child to the parent. We first show that T_{parent} is acyclic.
Let $i = parent[j]$ in T_{parent} for some nodes i and j. Then,
$code[f[j]] = i$ (Using (R1))
$\Rightarrow (z[i] \neq 0) \wedge (f[j] \leq z[i])$ (Using (R4))
$\Rightarrow f[j] < f[i]$ (Using (R5) for i)
Applying this argument repeatedly shows that the ancestor of a node has a higher f value than the f value for the node itself. This implies that no node is an ancestor of itself and hence T_{parent} is acyclic.

Every node in T_{parent} has outdegree either 0 or 1 depending upon the validity of the *parent* variable. We now show that every node except P_n has a valid parent and P_n forms the root of the tree. For a node $P_i, i \neq n$,
$f[i] \neq n$ (Using (R3)(1))
$\Rightarrow 1 \leq parent[i] = code[f[i]] \leq n$ (Using (R2),(R1))
Since the graph T_{parent} is acyclic and every node except P_n has a valid parent, P_n is root of the tree.

The above theorem just ensures that the *parent* pointers form a spanning tree. It does not enforce any relationship between the structure of the tree formed by the *parent* pointers and the tree corresponding to *code*. The next theorem establishes this relationship. The proof for the theorem can be found in the technical report [14].

Theorem 2. *If code, parent, f and z satisfy constraint set $\mathcal{C}$, then parent forms a rooted spanning tree isomorphic to the tree generated by code.*

The above theorem suggests that there is a possibility that the tree formed by *parent* is not same as the tree generated by *code*. For example, consider the value of the variables given in Table 4. It can be easily verified that these values satisfy the constraint set $\mathcal{C}$. The tree corresponding to *code* is the one we considered earlier in Figure 3. The tree generated by *parent* is shown in Figure 5. The two trees are not the same but they are isomorphic.

5 Maintaining Constraints

Each node i maintains $parent[i]$, $code[i]$, $f[i]$ and $z[i]$ and cooperates to ensure that the required constraints are satisfied, resulting in a valid rooted spanning tree. We present a strategy for efficient detection and correction of faults for each of the constraints. We will first discuss (R3) as it turns out to be most difficult to detect and correct.

5.1 Constraint (R3)

Constraint (R3)(1) is a local constraint which can be checked easily. Violation of this constraint can be fixed by simply setting f to a random number between 1 and $n-1$. Constraint (R3)(2) requires f to be a permutation on $1 \ldots n$. This can, in turn, be modeled in terms of the following constraints:
(C1) $\forall i : 1 \leq f[i] \leq n$ (C2) $\forall i, j : f[i] \neq f[j]$

The violation of (C1) is easy to detect. Every node i checks the value $f[i]$ periodically. If it is not between 1 and n, then a fault has occurred. The constraint (C2) is more interesting. At first glance it seems counter-intuitive that we can detect violation of (C2) in $O(1)$ messages. However, by adding auxiliary variables, the above task can indeed be accomplished. We maintain $g[i]$ at each process P_i such that, in a legal global state $f[i] = j \equiv g[j] = i$. Thus, g represents the

inverse of the array f. Note that the inverse of a function exists iff it is one-one and onto which is true in this case. If each process P_i maintains $f[i]$ and $g[i]$, then it is sufficient for a node to check periodically the following constraints:
(D1) $\forall i : 1 \leq f[i] \leq n$ (D2) $\forall i : 1 \leq g[i] \leq n$ (D3) $g[f[i]] = i$

It is easy to show that (C2) is implied by (D1)-(D3). If for some distinct i and j, $f[i]$ is equal to $f[j]$, then $g[f[i]]$ and $g[f[j]]$ are also equal. This means that $(g[f[i]] = i)$ and $(g[f[j]] = j)$ cannot be true simultaneously. (D3) can be checked by P_i by sending a message to $P_{f[i]}$ periodically, prompting $P_{f[i]}$ to check whether $g[f[i]] = i$ is true. Note that by introducing additional variables we have also introduced additional sources of data faults. It may happen that requirements (C1)-(C2) are met, but due to faults in g, constraints (D1)-(D3) are not met. We believe that the advantage of local detection of a fault outweighs this disadvantage.

The above scheme has an additional attractive property: If we assume that there is a single fault in f or g, then it can also be automatically corrected. The details for this scheme are given in the technical report [14].

5.2 Other Constraints

Constraints (R1), (R2) and (R5). Constraint (R1) is trivial to check locally. Each node i inquires node $j = f[i]$ for $code[j]$. If this value does not match $parent[i]$, then the constraint ($R1$) is violated. On violation, (R1) can be ensured by setting $parent[i]$ to $code[j]$. Constraint (R2) is also trivial to check and correct locally. Similarly, violation of (R5) can be detected easily and on a fault, $f[i]$ can be set to $z[i] + 1$.

Constraint (R4): This constraint can be modeled in terms of the following constraints:
(E1) $\forall i : (z[i] \neq 0) \Rightarrow (code[z[i]] = i)$ (E2) $\forall i, j : (code[j] = i) \Rightarrow (z[i] \geq j)$

For checking (E1), node i prompts the node $z[i]$ to verify that $code[z[i]] = i$. If the check fails, then $z[i]$ can be set to 0, which may not be the correct value for $z[i]$. If $z[i]$ is set incorrectly to 0, then constraint (E2) is also violated. As a result, while checking for (E2), $z[i]$ is set appropriately. For checking (E2), every node j sends a message to node $code[j]$ to verify that $z[code[j]] \geq j$. If (E2) is found to be violated upon receiving a message from node j, then $z[code[j]]$ is set to j.

5.3 Complete Algorithm

Depending upon the set of constraints ($\mathcal{R}$ or $\mathcal{C}$) that a process obeys, we have two versions of the algorithm. They differ in the guarantees about the resulting tree and their time complexities.

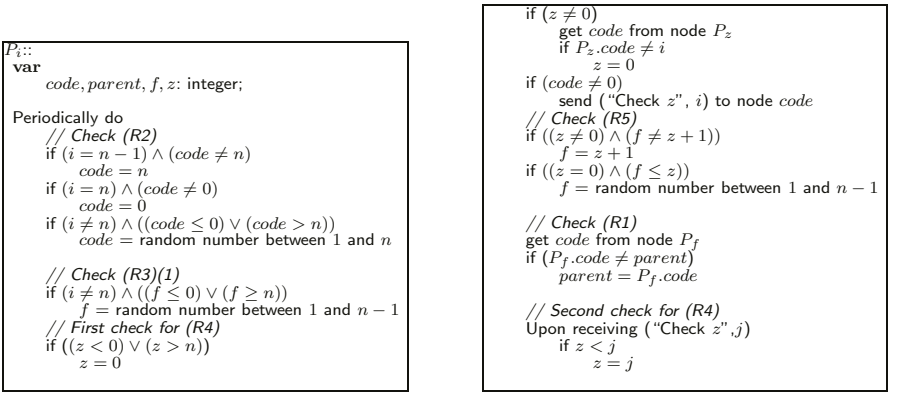

Fig. 6. Algorithm *SSR* for maintaining the constraint set $\mathcal{R}$

Maintaining $\mathcal{R}$. As we proved in Theorem 1, the set of constraints $\mathcal{R}$ is sufficient to maintain a spanning tree. The complete algorithm for process i to maintain the constraint set $\mathcal{R}$ is given in the Figure 6. We refer to this algorithm as *SSR*. In the algorithm, instead of denoting variables like $code[i]$, we have used $P_i.code$ to emphasize that the variables are local to the processes and are not shared. The algorithm checks the constraints one by one and on the violation of a constraint, it takes corrective action. For checking constraints which involve obtaining the value of another process's variable, we have used a primitive *get*. This involves the sender sending a request for the required variable and the receiver then replying with the appropriate value. A separate thread would be used by a process to respond to the *get* requests from other processes. Another point to notice in the algorithm is the asynchronous receive of the "Check z" messages. These messages would be received by a third thread which is woken up whenever a message arrives. Our system model takes this into account by assuming that a process alternates between the three threads of execution. The formal proof of correctness of the algorithm is given in the technical report [14].

At this point, we also give our reasons for choosing a different model for evaluation of our strategy. In the previous works, the *asynchronous rounds* [9, 12] model was used. The first asynchronous round in an execution E is the shortest prefix E' of E such that each process executes at least one step in E'. Let E'' be the suffix of E that follows E'. The second round of E is the first round of E'', and so on. The stabilization time of an algorithm is the maximum number of rounds it executes before the system reaches a legal state. In this model, a process waiting for a message receives the message in one round whereas if the message receive is asynchronous, it fails to provide any guarantees. In practice, running time of both the algorithms depends upon the message delivery time in a similar way and hence their time complexities should be comparable. We try to achieve this by putting a bound on the message delivery time. Our algorithm, like most other self-stabilizing algorithms, is structured as a loop that is executed periodically. We refer to this loop as a *cycle*.

The following theorems give the time and message complexity of this algorithm averaged over all the nodes.

Theorem 3. *The algorithm SSR requires $O(1)$ time per node and $O(1)$ messages per node on average in one asynchronous cycle with each message of size $O(\log n)$.*

Proof. In the algorithm *SSR*, every process sends a constant number of *get* requests and one "Check z" request. This results in a total of $O(n)$ messages. Corresponding to the *get* requests, there would be a total of $O(n)$ replies. The number of "Check z" messages received by a process i depends upon the number of times i appears in *code*. Assuming a random code, every node processes $O(1)$ messages on average. Since each node takes constant number of steps in an asynchronous cycle, every process requires $O(1)$ time on average to complete one asynchronous cycle. Moreover, since each message sends an *id* between 1 and n, each message is of size at most $O(\log n)$.

The following theorem gives the stabilization time of the algorithm in terms of our model. The proof for the theorem is given in the full version of the paper [14].

Theorem 4. *[14] The algorithm SSR stabilizes in $O(d)$ time, where d is the upper bound on the number of times a node appears in code.*

The problem of choosing the first $n-2$ numbers of *code* at random can be considered as the problem of randomly assigning $n-2$ balls to n bins. The following theorem is a standard result in probability theory [17][Theorem 3.1]:

Theorem 5. *If n balls are thrown randomly in n bins, then with the probability at least $1-\frac{1}{n}$, no bin has more than $\frac{e \log n}{\log \log n}$ balls.*

For a randomly chosen code, this theorem provides an upper bound for d and hence an upper bound on the stabilization time with very high probability.

These results show that the set $\mathcal{R}$ of constraints can be maintained efficiently. The algorithm for maintaining the constraint set $\mathcal{C}$, called *SSC*, is given in the technical report [14]. The *SSC* algorithm can take upto $O(n)$ time for stabilization.

5.4 Changing the Root Node

The algorithms *SSR* and *SSC* can be easily modified to allow the root node to change dynamically i.e. any node (not necessarily n) can become the root of the tree and the root can be changed during the operation of the algorithm. This can be achieved by changing the constraints (R2) and (R3)(1) in the following way:
(R2) $(\forall i : 1 \leq i \leq n-1 \Rightarrow 1 \leq code[i] \leq n) \wedge (code[n] = 0)$
(R3)(1) $\forall i : i \neq code[n-1] \Rightarrow 1 \leq f[i] < n$

The modified constraints are also easy to check and maintain. In the next section we present an application which utilizes this feature.

5.5 Systematically Changing the Tree

The SSR algorithm ensures that if the code is changed, then the spanning tree stabilizes to reflect that change. This property of the algorithm could be used by an application to purposefully change the spanning tree. If we are maintaining the set of constraints $\mathcal{R}$, then changing the code value at a node may not always result in a change in the tree. To get around this problem, whenever a node i wishes to change the tree, it changes the value of $code[f[i]]$ by requesting node $f[i]$. This changes $parent[i] = code[f[i]]$ and hence the spanning tree changes. Additionally, this may result in some more changes in the spanning tree as the parent of some other nodes may also get modified. This technique could be useful for load balancing purposes.

6 Conclusion and Future Work

In this paper we presented a new technique for maintaining spanning trees using labeled tree encoding. Our method requires $O(1)$ messages per node on average in one cycle and provides fast stabilization. It also offers a method for changing the root of the tree dynamically and systematically changing the tree for load balancing purposes. This work also demonstrates the use of the concept of *core* and *non-core* states for designing self-stabilizing algorithms.

It would be interesting to extend this work for general topology. Another research direction would be to modify the algorithm so that it does not require the nodes to have labels from 1 to n.

References

1. Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *Proc. of the 4th Int'l Workshop on Distributed Algorithms*, pages 15–28. Springer-Verlag, 1991.
2. S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithm. In *Proc. of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 400–410, 1993.
3. G. Antonoiu and P. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *European Conference on Parallel Processing*, pages 480–487, 1997.
4. A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.
5. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.
6. Z. Collin and S. Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.
7. N. Deo and P. Micikevicius. Prufer-like codes for labeled trees. *Congressus Numerantium*, 151:65–73, 2001.
8. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

9. S. Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, 2000.
10. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *MCC Workshop on Self-Stabilizing Systems*, 1989.
11. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. of the ninth annual ACM symposium on Principles of Distributed Computing*, pages 103–117. ACM Press, 1990.
12. S. Dolev, A. Israeli, and S. Moran. Uniform self-stabilizing leader election. In *Proc. of the 5th Workshop on Distributed Algorithms*, pages 167–180, 1991.
13. F. C. Gaertner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, EPFL, Oct 2003.
14. V. K. Garg and A. Agarwal. Self-stabilizing spanning tree algorithm with a new design methodology. Technical report, University of Texas at Austin, 2004. Available as `"http://maple.ece.utexas.edu/TechReports/2004/TR-PDS-2004-001.ps"`.
15. S. Huang and N. Chen. A self stabilizing algorithm for constructing breadth first trees. *Information Processing Letters*, 41:109–117, 1992.
16. C. Johnen. Memory efficient, self-stabilizing algorithm to construct bfs spanning trees. In *Proc. of the sixteenth annual ACM symposium on Principles of Distributed Computing*, page 288. ACM Press, 1997.
17. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
18. E. H. Neville. The codifying of tree-structure. *Proceedings of Cambridge Philosophical Society*, 49:381–385, 1953.