A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms

Felix C. Gärtner

Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Distributed Programming Laboratory, CH-1015 Lausanne, Switzerland, fcg@acm.org

Swiss Federal Institute of Technology (EPFL)
School of Computer and Communication Sciences
Technical Report IC/2003/38
June 10, 2003

Abstract. Self-stabilizing systems can automatically recover from arbitrary state perturbations in finite time. They are therefore well-suited for dynamic, failure prone environments. Spanning-tree construction in distributed systems is a fundamental task which forms the basis for many other network algorithms (like token circulation or routing). This paper surveys self-stabilizing algorithms that construct a spanning tree within a network of processing entities. Lower bounds and related work are also discussed.

1 Introduction

Imagine what happens if you take an arbitrary distributed algorithm, e.g., for termination detection, and start it in a state where one of its variables has been set to a random value from its domain. Usually, the behavior is not predictable: either the algorithm will output garbage (e.g., declare a computation as finished although it is still running), or (most probably) it will deadlock (e.g., it will fail to output anything at all). It may be argued, that changing the value of a variable is unfair: no algorithm can tolerate such manipulations since algorithms have to rely on proper initialization. This argument, however, is not true.

Self-stabilizing algorithms [19,22] are guaranteed to recover from an arbitrary perturbation of their local state in a finite number of execution steps. This means that the variables of such algorithms do not need to be initialized properly. Assign to each variable (even the program counter) an arbitrary value from its domain and the algorithm will eventually start to behave as expected. Arbitrary state perturbations can also happen without curious users playing around with their algorithm: Cosmic rays in spacecraft for example can arbitrarily change the contents of memory cells in random access memory to a certain extent. Thus, self-stabilizing algorithms have the desirable property to recover from such faults automatically.

In distributed systems, a *spanning tree* is the basis for many complex distributed protocols. To define a spanning tree, the network is formalized as a graph $G = (V, E)$ where V is the set of network nodes (vertices) and E is the set of communication links (edges) between network nodes (formally it is a relation over V, i.e., a subset of $E \times E$). A spanning tree $T = (V, E')$ of G is a graph consisting of the same set of nodes V, but only a subset $E' \subseteq E$ of edges such that there exists exactly one path between every pair of network nodes (see Fig. 1). Basically, this means that the graph is connected (there is at least one path between any two nodes) and it does not contain cycles (there is at most one path between any two nodes). One of the basic theorems of spanning trees states that in a network of n nodes, the tree contains exactly $n-1$ communication links. A spanning tree is in general not unique.

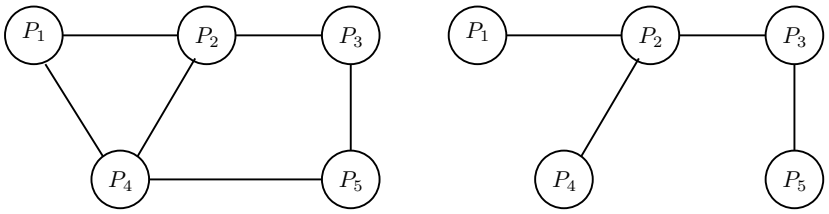

Fig. 1. Example of a network of five nodes (left) and a spanning tree of the network (right).

A spanning tree in the network is often a prerequisite for more involved network protocols like routing or token circulation. It can also increase the efficiency of network protocols. Take for example the problem of broadcasting messages in the network. There are algorithms which "flood" the network, i.e., the broadcast message is sent to *all* neighbors. Consequently, the message crosses all communication links before the protocol has finished. However, if a spanning tree of the network is available, the message only needs to be sent to all those nodes which are neighbors in the spanning tree. Instead of crossing all E links, it just crosses $n-1$ links, and since $|E|$ is usually significantly larger than $n-1$ a spanning tree can considerably reduce the message complexity of the broadcast algorithm.

In this paper, we survey self-stabilizing distributed algorithms which construct a spanning tree in a distributed system. These algorithms solve a basic problem in a very robust way and can be used as building blocks in fault-tolerant and dynamic applications. Apart from collecting and organizing the references to the literature, this survey also aims at aiding system designers in the choice of which algorithm to apply in which setting. This survey tries to be complete in its reference list, but this goal cannot be claimed because spanning tree construction is such a basic task that it frequently is handled as a subtask of other self-stabilizing protocols (see for example the paper by Arora and Gouda [11]) and so a treatment of the subject matter may be buried in any publication on self-stabilization. The author of this survey invites readers to send suggestions on

which other papers should be included in future revisions of this paper. Also, four papers have not been available to the author at the time of writing [24, 38–40]. Support in obtaining copies of these works is also very much appreciated.

The paper first gives some background about the different system assumptions made in the literature (Section 2) and recalls impossibility results and lower bounds (Section 3). We then present the most important basic algorithms and algorithmic ideas in a historical perspective (Section 4), followed by a roundup of other related work (Section 5).

2 System Assumptions

The system is usually modeled as a graph of processing elements (processors, processes, nodes), where the edges between these elements model unidirectional or bidirectional communication links. In this paper, we denote by n the number of nodes in the system and by N an upper bound on n. Communication network is usually restricted to the neighbors of a particular node. We denote by δ the diameter of the network (i.e., the length of the longest unique path between two nodes) and by Δ upper bound on δ. A network is static if the communication topology remains fixed. It is dynamic if links and network nodes can go down and recover later. In the context of dynamic systems, self-stabilization refers to the time after the "final" link or node failure. The term "final failure" is typical for the literature on self-stabilization: Because stabilization is only guaranteed *eventually*, the assumption that faults eventually stop to occur is an approximation of the fact that there are no faults in the system "long enough" for the system to stabilize. In any case, it is assumed that the topology remains connected, i.e., there exists a path between any two network nodes even if a certain number of nodes and links may crash.

Algorithms are modeled as state machines performing a sequence of steps. A step consists of reading input and the local state, then performing a state transition and writing output. Communication can be by exchanging messages over the communication channels. But the more common model for communication is that of shared memory or shared registers [22]. It assumes that two neighboring nodes have access to a common data structure, variable or register which can store a certain amount of information. These variables can be distinguished between input and output variables (depending on which process can modify them). When executing a step, a process may read all its input variables, perform a state transition and write all its output variables in a single atomic operation. This is called *composite atomicity* [27]. A weaker notion of a step (called *read/write atomicity* [27]) also exists where a process can only either read or write its communication variables in one atomic step. A related characteristic of a system model is its execution semantics. In the literature on self-stabilization this is encapsulated within the notion of a *scheduler* (or *daemon*) [19]. Under a central daemon, at most one processing element is allowed to take a step at the same time.

The individual processes can be *anonymous*, meaning that they are indistinguishable and all run the same algorithm. Often, anonymous networks are called *uniform* networks [27]. A network is *semi-uniform* if there is one process (the root) which executes a different algorithm [27]. While there is no way to distinguish nodes, in uniform or semi-uniform algorithms nodes usually have a means of distinguishing their neighbors by ordering the incoming communication links. In the most general case it is assumed that processes have globally unique identifiers.

An algorithm may be randomized, i.e., have access to a source of randomness (a random number generator or a random coin flip). If an algorithm is not randomized, we will call it deterministic.

Two kinds of spanning trees may be distinguished: *breadth-first search* (BFS) trees result from a breadth-first traversal of the underlying network topology [37]. Similarly, *depth-first search* (DFS) trees are obtained from a depth-first traversal. A notion underlying DFS and BFS trees is that of a *rooted tree*. A rooted spanning tree is a spanning tree of the network where the tree edges are consistently directed with respect to a particular node (the root). Edges can be directed towards the root or "away from" the root. Rooted spanning trees have a notion of "parent" and naturally result from the execution of semi-uniform algorithms. In fact, since almost all algorithms use a single pointer (to a neighbor, the parent) to store the structure of the tree, all these algorithms implicitly construct a rooted spanning tree.

3 Impossibilities and Lower Bounds

For the case of spanning-tree construction, Angluin [6] showed that it is impossible to deterministically construct a spanning tree in uniform networks. Intuitively, this is caused by problems of symmetry, and so at least a semi-uniform setting (e.g., a distinguished root processor) or a source of randomization is needed. This work does not refer to self-stabilizing algorithms, but it should be clear that the impossibility result also holds for self-stabilizing case since it is "harder" to solve a problem without being able to rely on an initial state.

The usual time-complexity measure for self-stabilizing algorithms is that of *rounds* [29]. In synchronous models algorithms execute in rounds, i.e., processors execute steps at the same time and at a constant rate. Rounds can be defined in asynchronous models too, where the first round ends in a computation when every processor has executed at least one step. In general, the i-th round ends, when every processor has executed at least i steps. In general, communication between any two processors in a particular system takes at least $\Omega(d)$ rounds. This is because it normally takes at least one round to propagate information between two adjacent processors. For the case of self-stabilizing spanning-tree construction and under certain assumptions, an arbitrary initial state may make it necessary to propagate information through the entire network. Therefore, a general lower bound of $\Omega(d)$ rounds can be assumed for self-stabilizing spanning-

tree algorithms. By combining the algorithm with a hierarchical structure and sacrificing true distribution, this bound can be lowered [28].

We are aware of only a single result on lower bounds regarding the space complexity (i.e., the amount of state necessary to perform self-stabilizing spanning tree construction). Dolev, Gouda, and Schneider [23] proved that self-stabilizing spanning tree construction needs at least $\Omega(\log n)$ bits per processor if the algorithm is *silent*, i.e., if the contents of the communication registers eventually stop changing. If the algorithm is not required to be silent, Johnen [35] showed that it is possible to construct an algorithm using only $O(1)$ bits per edge in a uniform rooted network with a central daemon. In an appendix to their paper, Itkis and Levin [34] did the same in anonymous networks using randomization.

4 Basic Algorithms

The first self-stabilizing spanning-tree constructions algorithms were published in the beginning of the 1990s. A couple of papers were published at that time which independently developed solutions for different network settings using similar algorithmic ideas.

4.1 The Algorithm by Dolev, Israeli and Moran

One of the first papers to appear was by Dolev, Israeli and Moran [25, 27] in 1990. It contains a self-stabilizing BFS spanning-tree construction algorithm for semi-uniform systems with a central daemon under read/write atomicity. In the algorithm, every node maintains two variables: (1) a pointer to one if its incoming edges (this information is kept in a bit associated with each communication register), and (2) an integer measuring the distance in hops to the root of the tree. The distinguished node in the network acts as the root. The algorithm works as follows: The network nodes periodically exchange their distance value with each other. After reading the distance values of all neighbors, a network node chooses the neighbor with minimum distance *dist* as its new parent. It then writes its own distance into its output registers, which is $dist + 1$. The distinguished root node does not read the distance values of its neighbors and simply always sends a value of 0.

The algorithm stabilizes starting from the root process. After sufficient activations of the root, it has written 0 values into all of its output variables. These values will not change anymore. Note that without a distinguished root process the distance values in all nodes would grow without bound. More specifically, after reading all neighbors values for k times, the distance value of a process is at least $k + 1$. This means, that after the root has written its output registers, the direct neighbors of the root—after inspecting their input variables—will see that the root node has the minimum distance of all other nodes (the other nodes have distance at least 1). Hence, all direct neighbors of the root will select the root as their parent and update their distance correctly to 1. This line of reasoning can

be continued incrementally for all other distances from the root. Hence, after $O(\delta)$ update cycles the entire tree will have stabilized.

The above algorithm is used by Dolev [21] as the basis for a topology update algorithm in dynamic networks. This is the first paper we are aware of which refers to the time optimality of the above algorithm.

Based on the same algorithmic idea, Collin and Dolev [18] present a semi-uniform spanning-tree algorithm under a central daemon and read/write atomicity that constructs a DFS tree (instead of a BFS tree). A similar algorithm which also constructs a DFS tree but uses composite atomicity. was published by Herman [31, Chapter 6] three years earlier.

In this algorithm, the outgoing links at every process are ordered, and the DFS tree is defined as the tree resulting from a DFS graph traversal always selecting the smallest outgoing edge. Instead of writing its current level into the output registers, it writes a representation of its current estimate of the *path* (the sequence of outgoing link identifiers) to the root. The root repeatedly writes the "empty path" $\perp$ to its output registers. If a node has k neighbors, there are k alternative paths to choose from. From these, the node chooses the path which is minimal according to a lexicographic order which prefers smaller link identifiers, For example, $(\perp) < (\perp, 1) < (\perp, 1, 1) < (\perp, 2) < (1)$ and so a node does not choose the shortest path to the root but along the smallest link identifiers. The authors remark that this is the same principle as used in the algorithm of Dolev, Israeli and Moran [25]. The memory requirements for the DFS algorithm however are $O(n \log K)$ bits where K is an upper bound on the maximum degree of a node. The time complexity is $O(\delta n K)$ rounds.

4.2 The Algorithm by Afek, Kutten and Yung

In the same year as Dolev, Israeli and Moran [25] published their algorithm, Afek, Kutten and Yung [3] presented an self-stabilizing algorithm for a slightly different setting. Their algorithm also constructs a BFS spanning-tree in the read/write atomicity model. However, they do not assume a distinguished root process. Instead they assume that all nodes have globally unique identifiers which can be totally ordered. The node with the largest identifier will eventually become the root of the tree.

The idea of the algorithm is as follows: Every node maintains a parent pointer and a distance variable like in the algorithm above, but it also stores the identifier of the root of the tree which it is supposed to be in. Periodically, nodes exchange this information. If a node notices that it has the maximum identifier in its neighborhood, it makes itself the root of its own tree. If it learns that there is a tree with a larger root identifier nearby, it joins this tree by sending a "join request" to the root of that tree and receiving a "grant" back. The subprotocol together with a combination of local consistency checks ensures that cycles and fake root identifiers are eventually detected and removed.

The algorithm stabilizes in $O(n^2)$ asynchronous rounds and needs $O(\log n)$ space per edge to store the process identifier. The authors argue this to be

optimal since message communication buffers usually communicate "at least" the identifier.

4.3 The Algorithm by Arora and Gouda

Also in 1990, Arora and Gouda [10,11] published a self-stabilizing BFS spanning-tree algorithm for the composite atomicity model under a central daemon. Similar to Afek, Kutten and Yung, they also assume unique identifiers and the node with maximum identifier eventually acts as the root of the system. In contrast to Afek, Kutten and Yung, the algorithm needs a bound N on the number n of nodes in the network to work correctly.

The bound N is necessary because the algorithm uses a different technique to detect and remove cycles. Again, every node maintains variables for distance, parent and root identifier. Periodically, every node compares its own distance and root identifier setting with the values stores in the node pointed to by the parent variable. In the "finished" spanning tree, the root identifiers should be the same and the distance should be the distance of the parent incremented by 1. If this is not the case, the root identifier is copied from the parent and the distance is set to the parent's distance plus 1. If there is a cycle in the tree (for example due to improper initialization), the distance values are incremented along this cycle without bound. Hence, a cycle is detected when the distance value supercedes the bound N. The first node to detect this makes itself the root of a new tree. A node also continuously monitors the root identifier and distance settings of its neighbors. If a neighbor has a larger root identifier or the same identifier with smaller distance, the node adjusts its values accordingly.

Knowledge of the bound N allows the algorithm of Arora and Gouda [11] to be simpler than the one by Afek, Kutten and Yung [3] but the stabilization time is $O(N^2)$, which can be much larger than $O(n^2)$. In dynamic networks where network nodes may go down, a stabilization time in the order of the actual number of nodes is preferable.

4.4 The Algorithms by Huang et al.

The same idea for cycle breaking (through "bumping up" the distance counter) was presented in 1991 by Chen, Yu and Huang [16]. In this paper, they present a self-stabilizing spanning tree algorithm for semi-uniform systems with composite atomicity. The fact that there is a distinguished root makes the algorithm even simpler than the one by Arora and Gouda [11]. However, the algorithm does not necessarily stabilize to a BFS tree since the choice of a new parent after breaking a cycle is non-deterministic (governed by the scheduler). This was adapted in a later paper by Huang and Chen [33] to yield an algorithm which constructs a BFS tree using knowledge of the size n of the network.

Interestingly, Huang and Chen [33] see the contribution of the latter algorithm in the technique to prove stabilization, not in the algorithm itself since the one by Dolev, Israeli and Moran [25] achieves the same goal but assumes read/write atomicity instead of composite atomicity. Four years after Chen, Yu

and Huang's paper [16], the algorithm was re-invented (with slight modifications) by Antonoiu and Srimani [7]. In this paper too, the authors claim that their proof technique is as important a contribution as the algorithm itself.

4.5 The Algorithm by Afek and Bremler

Afek and Bremler [1, 2] revisit the problem of self-stabilizing spanning-tree construction. They give an algorithm for systems with unidirectional, bounded capacity message passing links. They assume unique identifiers and give adoptions of the algorithm for the synchronous and the asynchronous network case. The network node with the minimum identifier eventually plays the role of the root in the spanning tree.

The algorithm exploits a new design idea called "power supply" which enables the algorithm to have unique features. For example, the algorithm stabilizes in $O(n)$ rounds without any knowledge about n. The power supply method exploits the fact that self-stabilizing algorithms must continuously check their own state. In the algorithm, nodes which are part of some spanning tree expect to receive "power" from the root of the tree (power means a continuous flow of certain messages, one per round). The idea of the algorithm is that only legal roots may be the source of power and that nodes attached to fake roots eventually fail to receive power and subsequently make themselves the root of a new tree. Whenever a node receives power from a neighbor with a smaller identifier, it attaches itself to its tree.

In the asynchronous case, the power supply idea is implemented using using different types of messages: *weak* messages are exchanged periodically between the nodes to synchronize their state, while *strong* messages carry power. Afek and Bremler give a generic power supply algorithm which can be instantiated to a leader election algorithm, or an algorithm to construct DFS or BFS spanning trees.

5 Other Related Work

In all of the above algorithms, processes have to maintain a variable measuring the distance from the root of the tree which must be communicated to the neighbors. This means that communication variables must have at least $O(\log n)$ bits. Work by Awerbuch and Ostrovsky [13] reduce this requirement to $O(\log^* n)$ bits per edge. As mentioned above, this memory requirement was reduced to $O(1)$ by Johnen [35].

For completeness, we briefly mention other papers which have investigated spanning-tree construction.

Awerbuch, Patt-Shamir and Varghese [14] give an algorithm based on unique identifiers within the paradigm of "local checking and correction". Aggarwal and Kutten [4, 5] give a spanning-tree algorithm for message passing environments which uses a clever data structure to allow stabilization with $O(\delta)$ time without any knowledge of network diameter or number of nodes. The basic version

of the algorithm was designed for anonymous networks and uses randomization to create unique identifiers. Awerbuch et al. [12] present another time-optimal spanning-tree algorithm which is based on unique identifiers, but it requires knowledge of a bound Δ on the network diameter. Similarly, Dolev, Israeli and Moran [20, 26] present an algorithm for anonymous networks which stabilizes in $O(\delta)$ rounds but requires knowledge of a bound N on network size. Other authors have investigated other flavors of spanning-tree construction (e.g., minimum diameter spanning tree [15] and minimum spanning tree [8,9,32] in a graph with weighted edges). Ghosh, Gupta and Pemmaraju [30] study *fault-containing* spanning-tree construction, meaning that the effects of faults on the algorithm (e.g., its stabilization time) depend on the severity of faults (e.g., the number of corrupted processes). Their work is based on the algorithm by Chen, Yu and Huang [16].

6 Conclusions

The spanning-tree algorithms mentioned in this paper have been applied in many different settings in practice. A striking example is the self-stabilizing file system developed by Dolev and Kat at Ben Gurion University in Israel [36] which uses a variant of the algorithm by Dolev, Israeli and Moran [27] to implement a reliable data storage subsystem. As another example, consider the protocol eliminating redundant paths in switched Ethernets [17]. If a network segment becomes unreachable or network parameters are changed, the protocol automatically reconfigures the spanning-tree topology by activating a standby path (if one exists). The protocol can be briefly described as follows: Initially, switches believe they are the root of the spanning tree but do not forward any packets. Governed by a timer, they regularly exchange status information. These messages contain (1) the identifier of the transmitting switch (usually a MAC address), (2) the identifier of the switch which is believed to be the root of the tree, and (3) the "cost" of the path towards the root. Using this information, a switch chooses the "shortest" path towards the root. If there are multiple possible roots, it selects the root with the smallest identifier (lowest MAC address). Links which are not included in the spanning tree are placed in blocking mode. Blocking links do not forward packets but still transport status information.

We expect to see many other applications, derivations and re-inventions of the algorithms presented in this paper in the future.

Acknowledgments

This paper evolved from an enjoyable collaboration with Henning Pagnia, the results of which were published at SSS 2003 [28]. Support by the Deutsche Forschungsgemeinschaft (DFG) as part of the Emmy Noether programme is also gratefully acknowledged.

References

1. Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power-supply (extended abstract). In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 111–120, New Orleans, Louisiana, 5–7 January 1997.
2. Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 1998(3), December 1998.
3. Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In Jan van Leeuwen and Nicola Santoro, editors, *Distributed Algorithms, 4th International Workshop*, volume 486 of *Lecture Notes in Computer Science*, pages 15–28, Bari, Italy, 24–26 September 1990. Springer, 1991.
4. S. Aggarwal. Time optimal self-stabilizing spanning tree algorithms. Technical Report MIT-LCS//MIT/LCS/TR-632, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1994.
5. S. Aggarwal and S. Kutten. Time optimal self–stabilizing spanning tree algorithms. In Rudrapatna K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 400–410, Berlin, Germany, December 1993. Springer-Verlag.
6. Dana Angluin. Local and global properties in networks of processors (extended abstract). In ACM, editor, *Conference proceedings of the twelfth annual ACM Symposium on Theory of Computing: papers presented at the symposium, Los Angeles, California, April 28–30, 1980*, pages 82–93, New York, NY, USA, 1980. ACM Press.
7. Gheorghe Antonoiu and Pradip K. Srimani. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers and Mathematics with Applications*, 30:1–7, 1995.
8. Gheorghe Antonoiu and Pradip K. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *Euro-Par'97 Parallel Processing, Proceedings*, number 1300 in Lecture Notes in Computer Science, pages 480–487. Springer-Verlag, 1997.
9. Gheorghe Antonoiu and Pradip K. Srimani. A self-stabilizing distributed algorithm for minimal spanning tree problem in a symmetric graph. *Computers and Mathematics with Applications*, 35(10):15–23, 1998.
10. Anish Arora and Mohamed Gouda. Distributed reset. In *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 472 in Lecture Notes in Computer Science, pages 316–333. Springer-Verlag, 1990.
11. Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, September 1994.
12. Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In ACM, editor, *Proceedings of the twenty-fifth annual ACM Symposium on the Theory of Computing, San Diego, California, May 16–18, 1993*, pages 652–661, New York, NY, USA, 1993. ACM Press.
13. Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network reset. In *Symposium on Principles of Distributed Computing (PODC '94)*, pages 254–263, New York, USA, August 1994. ACM Press.

14. Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.
15. Franck Butelle, Christian Lavault, and Marc Bui. A uniform self-stabilizing minimum diameter tree algorithm (extended abstract). In Jean-Michel Hélary and Michel Raynal, editors, *Distributed Algorithms, 9th International Workshop, WDAG '95*, volume 972 of *Lecture Notes in Computer Science*, pages 257–272, Le Mont-Saint-Michel, France, 13–15 September 1995. Springer-Verlag.
16. Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.
17. Cisco Systems Inc. Using VlanDirector system documentation. Internet: `http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/sw_ntman/cwsimain/cwsi2/cwsiug2/vlan2/index.htm`, 1998.
18. Zeev Collin and Shlomi Dolev. Self-stabilizing depth first search. *Information Processing Letters*, 49:297–301, 1994.
19. Edsger W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
20. S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.
21. Shlomi Dolev. Optimal time self stabilization in dynamic systems (preliminary version). In André Schiper, editor, *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG93)*, volume 725 of *Lecture Notes in Computer Science*, pages 160–173, Lausanne, Switzerland, 27–29 September 1993. Springer-Verlag.
22. Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
23. Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.
24. Shlomi Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.
25. Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In Cynthia Dwork, editor, *Proceedings of the 9th Annual ACM Symposium on Principles of Distribted Computing*, pages 103–118, Québec City, Québec, Canada, August 1990. ACM Press.
26. Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election (extended abstract). In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 167–180, Delphi, Greece, 7–9 October 1991. Springer, 1992.
27. Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
28. Felix C. Gärtner and Henning Pagnia. Time-efficient self-stabilizing algorithms through hierarchical structures. In *Proceedings of the 6th Symposium on Self-Stabilizing Systems*, Lecture Notes in Computer Science, San Francisco, June 2003. Springer-Verlag.
29. C. Génolini and S. Tixeuil. A lower bound on dynamic k-stabilization in asynchronous systems. In *SRDS 2002 21st Symposium on Reliable Distributed Systems, IEEE Computer Society Press*, pages 211–221, 2002.

30. Sukumar Ghosh, A. Gupta, and S.V. Pemmaraju. A fault-containing self-stabilizing algorithm for spanning trees. *Journal of Computing and Information*, 2:322–338, 1996.
31. Ted Herman. *Adaptivity through distributed convergence*. PhD thesis, Department of Computer Science, University of Texas at Austin, 1991.
32. Lisa Higham and Zhiying Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*, number 2180 in Lecture Notes in Computer Science, Lisbon, Portugal, October 2001. Springer-Verlag.
33. Shing Tsaan Huang and Nian Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, February 1992.
34. G. Itkis and L. Levin. Fast and lean self-stabilizing asynchronous protocols. In Shafi Goldwasser, editor, *Proceedings: 35th Annual Symposium on Foundations of Computer Science, November 20–22, 1994, Santa Fe, New Mexico*, pages 226–239, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.
35. Colette Johnen. Memory efficient, self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC '97)*, pages 288–288, New York, August 1997. Association for Computing Machinery. Extended version appeared in the Proceedings of the third Workshop on Self-Stabilizing System (WSS'97), pages 125–140, Santa Barbara, CA, USA, August, 1997.
36. Ronen Kat. Self-stabilizing replication file system. Internet: `http://www.cs.bgu.ac.il/~srfs/`, September 2002.
37. Donald E. Knuth. *The Art of Computer Programming*, volume III (Sorting and Searching). Addison-Wesley, Reading, MA, second edition, 1997.
38. Reuay-Ching Pan, Jone-Zen Wang, and Louis R. Chow. A self-stabilizing distributed spanning tree construction algorithm with a distributed demon. *Tamsui Oxford Journal of Mathematical Sciences*, 15:23–32, 1999.
39. S. Sur and P. K. Srimani. A self-stabilizing distributed algorithm to construct BFS spanning trees of a symmetric graph. *Parallel Processing Letters*, 2(2-3):171–179, September 1992.
40. Ming-Shin Tsai and Shing-Tsaan Huang. A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters*, 4(1-2):65–72, June 1994.