A Maximum Degree Self-Stabilizing Spanning Tree Algorithm

Deniz Cokuslu[1,2,3]
denizcokuslu@iyte.edu.tr

Kayhan Erciyes[4]
kayhan.erciyes@izmir.edu.tr

Abdelkader Hameurlain[3]
hameur@irit.fr

[1]Izmir Institute of Technology, Department of Computer Engineering
Gulbahce, Urla, 35430 Izmir, Turkey
[2]International Computer Institute, Ege University
Bornova, 35100 Izmir, Turkey
[3]IRIT, Paul Sabatier University
118 Route de Narbonne, 31062 Toulouse, France
[4]Izmir University, Gursel Aksel Bulvari,
Uckuyular, 35350 Izmir, Turkey

Abstract

Spanning trees are fundamental topological structures in distributed environments which ease many applications that require frequent communication between nodes. Many different approaches are proposed in the literature for building spanning trees. Moreover, many other studies also consider fault tolerance and self stabilization properties while building these topological structures. Although there are known advantages of the self stabilization paradigm such as fault tolerance, tradeoffs between classical and self-stabilizing approaches to build spanning trees using distributed algorithms is not studied adequately. In this paper, we examine and compare two similar existing spanning tree construction algorithms which rely on different paradigms: classical versus self stabilization approach and propose a new self-stabilizing spanning tree construction algorithm which uses maximum degree heuristic while choosing the root node. We show experimentally that our new algorithm provides smaller tree diameters than the two existing approaches with favorable run-times.

1. Introduction

Spanning tree algorithms are widely used in many distributed applications. A spanning tree is a subset *S* of a graph *G* which contains every node in G and which does not contain any cycles. With the growth in the scale of distributed systems such as grid systems, the need for such topological control mechanisms has gained significant importance. These control mechanisms decrease the complexity of distributed algorithms caused by the connectivity of the underlying graph. By using spanning trees, many distributed applications can be implemented efficiently by making use of their properties. Especially, algorithms which involve multicast or broadcast operations exploit spanning trees since many efficient specific routing and multicasting algorithms exist. Distributed spanning tree construction algorithms have gained significant importance in the last decade since in environments such as grid systems, it is very hard to keep system wide information to build such topologies centrally. Many algorithms have been developed to build different types of spanning trees. Several studies focus on constructing minimum spanning trees in which the sum of edge weights is minimized [4, 16, 2, 10]. Minimum spanning trees are useful especially if communication costs are required to be minimized. Studies such as [6] construct a minimum degree spanning tree in which degrees of vertices are minimized. This property helps efficient routing of messages under heavy communication traffic. Some other studies aim to construct minimum diameter spanning tree in which diameter of the resulting spanning tree is minimized [7]. This property helps broadcasting of messages by optimizing the distance between vertices in the graph.

Besides classical distributed spanning tree construction algorithms, some of these studies also consider dynamicity and fault tolerance in their design. To cope up with the dynamicity of grid, self-stabilizing spanning tree algorithms have received attention recently [11]. Self-stabilizing paradigm ensures the validity of spanning tree structure without having the need to regenerate the spanning tree every time dynamicity occurs in the network. Self stabilization is a paradigm for distributed systems that allows the system to achieve a desired global state, even in the presence of faults. The concept of self stabilization was introduced in 1974 by Dijkstra [8]. The idea of self-stabilizing algorithms is that independently of the global state of the system, after a finite amount of time the system will reach to a correct global state. In a self-stabilizing algorithm, each node maintains local variables, and changes its state according to only on its local variables and the contents of its neighbors' local variables. The contents of a node's local variables constitute its local state and the union of all local states constitutes the system wide global state. Self stabilization is a very useful approach for the systems in which dynamicity occurs frequently, such as grid systems. Therefore it may be convenient to use self stabilization in such environments when designing distributed applications.

In this paper, we aim to show how self stabilization property affects the performance of a spanning tree construction algorithm. For this purpose, for the sake of simplicity we selected two spanning tree construction algorithms which do not consider any complex spanning tree property, a classical [15] and a self-stabilizing spanning tree construction algorithm [1]. We have examined, analyzed, implemented and tested these two algorithms, and compared the test results in terms of runtime of the algorithms and resulting spanning tree diameters. While runtime results show the tradeoff between the two approaches, the diameter results ensures that the algorithms result in spanning trees with similar characteristics. We also propose a new self-stabilizing spanning tree construction algorithm based on [1] which considers degrees of nodes in determining the root node of the resulting spanning tree. This heuristic is based on the observation that a root node will be involved in more frequent communications than the rest therefore it would be sensible that this node should have the property of having a higher degree than the average. We show that the resulting algorithm provides trees with the smaller diameters than the other two algorithms and also has favorable execution times experimentally. The rest of this paper is organized as follows: Section 2 gives background about the recent spanning tree construction algorithms. Section 3 examines the two selected spanning tree algorithms by giving detailed analysis. The new self-stabilizing spanning tree algorithm is described in Section 4 and the implementation details of the three algorithms is given in Section 5. Finally in Conclusion, the tradeoffs and comparisons and advantages of the new algorithm are examined.

2. Background

2.1. Self-stabilizing spanning tree algorithms

In [14], Kotowski and Kuszner proposed two self-stabilizing algorithms to find spanning tree in a polynomial number of rounds. In their study, they have distinguished a root node for the spanning tree construction. In their first algorithm, each node chooses the minimum id neighbor as its parent node. The root node has always id 0. In their second algorithm, each node holds a variable and the nodes use this variable to choose their parents instead of id numbers.

In [3], Antonoiu and Srimani proposed a self-stabilizing minimum spanning tree construction algorithm. In their design they assume that the edges in the graph have unique weights. They defined a reference node, root node, which is assumed to be privileged only once and can never be privileged again after it takes action. All other nodes take action according to the path specification between them and the root node. The difference between the rules for the root node and other nodes makes this algorithm a semi-uniform algorithm. At the end of the execution, a minimum spanning tree is constructed.

Gupta and Srimani proposed two self-stabilizing spanning tree algorithms in [12]. They consider ad-hoc networks as the system model in their design. In their first algorithm they construct s shortest path spanning tree in the ad-hoc network, while in their second algorithm, they built a minimum spanning tree. Both of those algorithms rely on the existence of a specified root node. The root node is assumed to be stable in the sense that it won't leave the system.

Blin et al. proposed a self-stabilizing algorithm to find minimum degree spanning tree in a network in [6]. They take the study proposed in [1] as a base and they improve the algorithm by adding a degree reduction module which decreases the degree of the resulting spanning tree in each round. As in [1], this algorithm is also prone to the failure of the root node.

Butelle et al. presented a uniform self-stabilizing algorithm which finds a minimum diameter spanning tree of an arbitrary positively real-weighted graph [7].

They have designed their algorithm to consist of two phases. In the first phase, a uniform randomized stabilizing unique naming protocol is designed in order to break the symmetry in the graph. In the second phase, they developed a self-stabilizing minimum diameter spanning tree protocol. Their algorithm relies on a center node in the network. The center node is used as the root node of the tree, and is determined in a self-stabilizing fashion.

In [13], Herault et al. proposed a self-stabilizing spanning tree algorithm for large scale systems. They use only two assumptions in their design. First assumption is that each node is equipped with a service that keeps a list of the node's neighbors. The second assumption is the existence of a failure detection service. By taking these assumptions into consideration, they proposed an algorithm in which each node starts being its own root. The roots try to merge their trees by asking each other to join by looking and comparing their identifiers. At the end of the algorithm, the biggest id node becomes the root of the final tree.

Pan et al. proposed a self-stabilizing spanning tree construction algorithm based on a self-stabilizing maximum finding method [17]. In their algorithm they find the maximum identifier and determine distances of each node to the maximum identifier node. They insert parent relations according to the distances of the nodes. At the end, the algorithm finds a BFS spanning tree. Since the algorithm does not require having a root node initially, it is prone to root node dynamicity.

Baala et al. presented a random walk based spanning tree construction algorithm which is self-stabilizing [5]. Their algorithm is based on random walk strategy which is executed by independent mobile agents. The agents are merged into a spanning tree when they meet each other regarding their color values. The final merged spanning tree constitutes a coherent spanning tree of the whole system. This algorithm is a uniform algorithm in which all nodes run the same algorithm. The root node is determined at the end of the algorithm, therefore this system does not require having a predetermined special root node. It is prone to node failures including the failure of the root node.

Dolev et al. proposed a uniform BFS spanning tree algorithm [9]. In their study, each node initiates the algorithm using its neighborhood. If the node's identifier is the greatest in its neighborhood, it selects itself as the root node. If it learns that there is a tree with a higher root identifier, the node joins to that tree by sending a joining request. At the end, the algorithm stabilizes and a BFS spanning tree is constructed.

Afek et al. proposed memory-efficient self-stabilizing spanning tree algorithm for general networks [1]. In their paper they also consider the dynamicity of the root node. In their design, every node starts to create a tree rooted by it. Then trees are merged by taking the biggest id root as the new root

2.2. Classical distributed spanning tree algorithms

Kshemkalyani and Singhal examined three different types of classical distributed spanning tree algorithms [15]: synchronous single initiator spanning tree algorithm, asynchronous single initiator spanning tree algorithm and asynchronous concurrent initiator spanning tree algorithm. In synchronous single initiator algorithm, authors assume existence of a root node which initiates the algorithm. They also assume that the algorithm executes in rounds. The algorithm is based on the flooding of a query message. The root node starts flooding, in each round a node sets the first sender of the message as its parent, if multiple messages are received in the same round, a random sender is selected as the parent. In asynchronous single-initiator algorithm, the same idea is used, but instead of synchronous rounds, the algorithm uses accept or reject messages to provide synchronization between nodes. In the third algorithm, called asynchronous concurrent initiator spanning tree algorithm, each node starts to create a spanning tree rooted by itself by sending query messages to their neighbors. In every received message, the node checks if the sender of the message has a bigger id than the current root node. If this is the case, the node joins that tree and sends a message to its neighbor indicating its new parent and root. Else, the node sends a reject message to the sender of the query, and continues its operation.

Many other techniques and algorithms exist to generate spanning trees in graphs. In this paper, we only focus on simple classical spanning tree algorithms in which no special constraints exist.

3. Main Algorithms

In this section, we examine, analyze, implement and compare the two spanning tree construction algorithms: memory-efficient self-stabilizing spanning tree algorithm [1] and asynchronous concurrent initiator spanning tree algorithm [15]. The main difference between these two algorithms is the self stabilization property. We aim to examine the influence of the self stabilization property on the performance of spanning tree construction.

3.1. Memory-efficient self-stabilizing spanning tree algorithm

In [1], authors propose a self-stabilizing spanning tree construction algorithm. They assume that nodes have unique identifiers and every node knows its neighbors. They also assume that nodes are aware of their neighbors' states; in other words, when a node fails, neighbors of the failed node notice this failure and update their neighbor lists. In this model, every node runs the same algorithm. At the beginning, each node tries to construct a spanning tree rooted at itself. Then the independent trees merge with each other considering the id of their roots. The larger process id overruns the process of lower id nodes. At the end, the biggest id node overruns all the remaining processes and becomes the root of the final spanning tree.

Each node *i* has local variables indicating its neighborhood (N_i), its parent node (P_i), its root node (R_i) and its distance to the root node (D_i). In the global legal state, each node has the same root with the biggest node id in the graph, parents of nodes are within their neighborhood and distance of each node is 1 bigger than its parent's distance ($D_i = D_{parent} + 1$). The root node has distance 0, and points to itself as its root and its parent. The algorithm runs in an infinite loop and checks the legal state conditions continuously. To achieve self stabilization, each node compares its neighbors' roots with its root node. If any neighbor has a bigger id root, then the node sends a request message to the bigger id root through its neighbor in order to join its tree. When a root node receives such a message it replies with a grant message and allows that node to join its tree.

The algorithm checks two conditions to determine whether it is in a legal state or not:

A: $[(R_i = i) \wedge (P_i = i) \wedge (D_i = 0)] \vee [(R_i > i) \wedge (P_i \in N_i) \wedge (R_i = R_{parent}) \wedge (\text{Distance} = D_{parent} + 1 > 0)]$

B: $A \wedge (R_i >= \max(R_{Ni}))$

Condition A states that either the node is a root node; or the node is not a root node, its root is bigger than its id and same as its parent's root, its parent is in its neighborhood and its distance to the root is 1 bigger than its parent's distance. Condition B states that condition A is true and node's root is the biggest among its neighbors' roots. When both A and B are true, the node is considered to be in the legal state. The algorithm checks these two conditions and takes action according to their correctness. When both conditions are false, then the node is in an illegal state and sets itself as the root node of its own tree. If condition A is true and node's root is not the biggest id node within its neighbors' roots, then the node chooses to join to the tree with bigger root id. To realize this, the node sends a request message to the biggest id root within its neighbors' roots through its neighbors. If the condition B is true, then the node is in the legal state. The algorithm also takes other actions in order to relay messages while checking these conditions. In case of failure of an inner node in the tree, the children of the failed node notice this failure and switch to the illegal state because of the failure of condition A. This failure triggers self stability property of this algorithm and relevant nodes take action in order to stabilize the spanning tree.

3.2 Asynchronous Concurrent Initiator Spanning Tree Algorithm

The asynchronous concurrent initiator spanning tree algorithm [15] is a basic classical distributed spanning tree construction algorithm in which nodes only need to know their neighborhood information. The algorithm does not ensure self stability property. It uses flooding in order to disseminate tree information to neighbors. Each node starts to build its own tree rooted at itself at the beginning. When a node wants to initiate the algorithm as a root, then it sends a query message to its neighborhood indicating that it is a root node. When a node receives a query message it compares the id of the sender with the id of its root, if new root has a bigger id then the node changes its root to the new root. In this case if the node is a leaf node then it sends an accept message, if not it sends a query message to its neighborhood indicating its new root. If new root id is smaller than current root of the node, then the node sends a reject message to the sender of the query message. When a node receives an accept message, then it adds the sender node to its children list. If that node is an initiator then it finishes its execution upon receiving accept messages from all its neighbors, if the node is a relay node, it sends accept message to its parent upon receiving accept messages from all its neighbors. When a node receives a reject message then it sends an accept message to its parent. More details about the algorithm can be seen in Fig.1.

```
1. If parent is undefined
2.      My parent = my id
3.      My root = my id
4.      Send QUERY message to all neighbors indicating that I am the root node

Upon receiving a message

5. If received message type is QUERY
6.      If my root < received root
7.          My parent = sender of the message
8.          My root = received root
9.          If I am a leaf node
10.             Send ACCEPT message to the sender of the message
11.         Else
12.             Send QUERY message indicating my root to my neighbors
13.     If my root > received root
14.         Send REJECT message to the sender

15. If received message type is ACCEPT
16.     If my root is same with the received root
17.         Add sender node to my children list
18.         If all my neighbors have replied to my QUERY
19.             If I am the root terminate
20.             Else send ACCEPT message to my parent

21. If received message type is REJECT
22.     If received root is same with my root
23.         If all my neighbors have replied to my QUERY
24.             If I am the root terminate
25.             Else send ACCEPT message to my parent
```

Fig.1 Asynchronous concurrent initiator spanning tree algorithm

4. Maximum Degree Self-Stabilizing Spanning Tree (MDST) Algorithm

We propose an extended version of memory-efficient self-stabilizing spanning tree algorithm which considers degrees of nodes while selecting the root node. In [1], the algorithm constructs the spanning tree according to the id of the nodes by choosing the biggest id node as the root node. This heuristic may have some disadvantages in complex networks because it does not consider the suitability of the chosen node as a root node. In many situations, predefined constraints are advantageous in choosing the root node. For instance, choosing the biggest degree node as the root, or choosing the center of the graph as the root node may be preferable if the desired operation is broadcasting or multicasting. On the other hand, simple and straightforward nature of memory-efficient self-stabilizing spanning tree algorithm makes it preferable because of its low complexity measures. For these reasons, we propose to modify the memory-efficient self-stabilizing spanning tree algorithm so that it constructs a spanning tree rooted at the biggest degree node. The assumption here is that the biggest id node in the graph is a better candidate to be a root node than a randomly chosen root, because this choice may decrease the diameter of the resulting spanning tree since this choice may decrease the height of the tree. To realize this, we first propose to use a simple hash function which combines degrees of nodes with their node ids. We called the resulting hash number as the *tag* of a node. This function inserts the degree of a node to the most significant part of the *tag*, and it inserts ids to the least significant part of the *tag* number. It normalizes the least significant part by inserting zeros at the beginning of the node ids. At the end, each node has a unique *tag* value which is sorted by nodes' degrees. The algorithm considers *tag* values in determining root of the nodes. The basic idea of our algorithm is the same with memory-efficient self-stabilizing spanning tree algorithm; the only difference is the usage of *tag* values instead of real ids of the nodes in determining root node. At the end of the execution of the MDST algorithm, a spanning tree rooted at the highest degree node is constructed.

5. Implementation and Experiments

We have implemented the three algorithms in the network simulator *ns2*. For the self-stabilizing algorithms, we have simulated failure detection module by using periodical messaging between neighboring nodes. Each node periodically sends a message to all its neighbors indicating its variables (root, parent and distance). If any message is not received from a neighbor in 2 periods then the node assumes that this neighbor has failed, and takes action. In both algorithms, we used UDP protocol for messaging. We did not use broadcast packet messages since *ns2* does not support broadcast messages in wired network scenarios.

We have generated 8 experiment scenarios by using randomly chosen wired network topologies ranging from 100 to 800 nodes. In self-stabilizing algorithms, the periods of status updates and self-stabilizing loops affect the runtime of the algorithm drastically. For that reason, we tried to choose the update interval as small as possible that *ns2* allows.

Runtime results of the algorithms can be seen in Fig.2. The difference between classical distributed approach and self-stabilizing approach is mainly caused by periodical updates in self-stabilizing algorithms. The effect of self stabilization property is magnified as the number of nodes is increased. While the runtime of asynchronous concurrent initiator spanning tree algorithm remains nearly constant, the runtime of self-stabilizing algorithms increases. But it is also seen that this increase is sub-linear with respect to the increase in the number of nodes which ensures the scalability of these algorithms.

The run-times of the algorithms with respect to the number of nodes is shown in Fig.2. It may be observed that MDST performs much better than the original self-stabilizing spanning tree algorithm. The difference is more evident for greater number of nodes. Fig.3 depicts the variation diameters of resulting spanning trees by using three different algorithms. The diameter of a tree is an important measure which is defined as the minimum distance between the two most distant nodes in the tree. It affects the performance of algorithms like routing and broadcasting algorithms, which use resulting spanning tree's properties. It may be seen that both memory-efficient self-stabilizing spanning tree algorithm and asynchronous concurrent initiator spanning tree algorithm have resulted in similar results in terms of tree diameters, while MDST algorithm has resulted in smaller diameter spanning trees. This difference is caused by the heuristic which is used in choosing the root node.

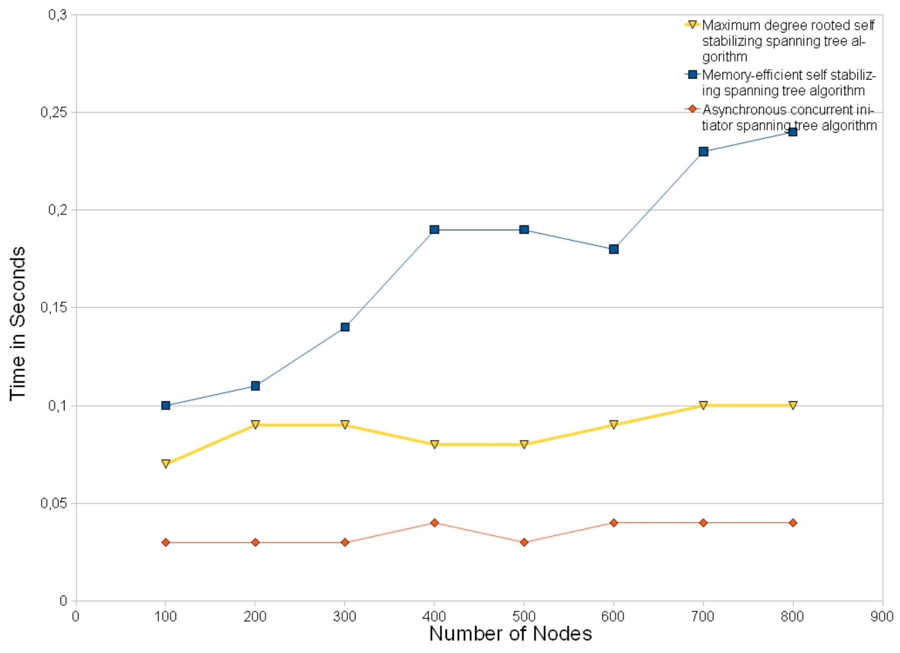

Fig.2 Runtime results of spanning tree construction algorithms

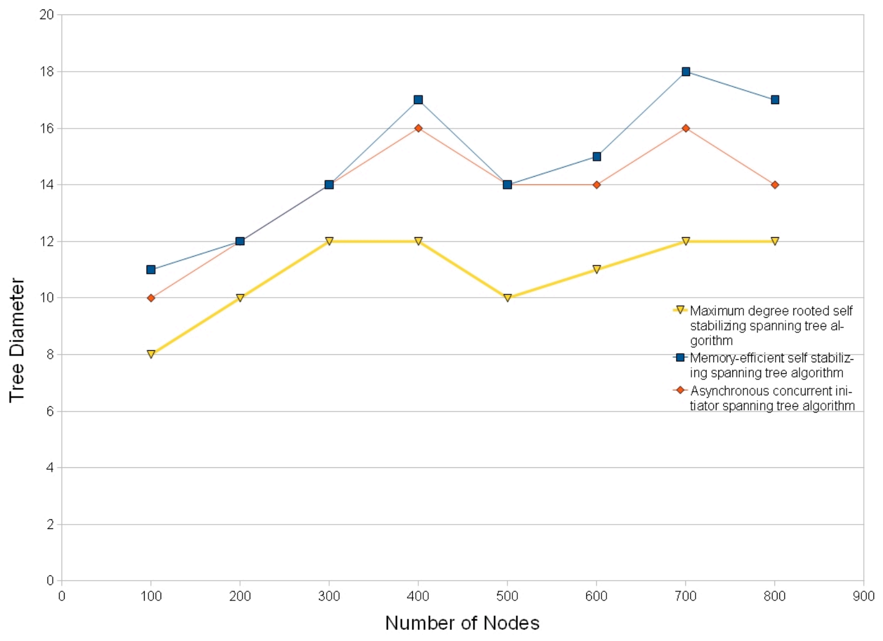

Fig.3 Diameter results of spanning tree construction algorithms

Conclusions

In this paper, we described, implemented and compared simulation results of two basic existing spanning tree construction algorithms which rely on different paradigms. We also proposed a new self-stabilizing spanning tree algorithm which relies on choosing the maximum degree node as the root. We showed the differences and similarities between self-stabilizing and classical approaches in terms of experiment results, and proposed a new algorithm. According to the implementation results we can say that both classical and self-stabilizing spanning tree algorithms behave similarly in terms of resulting spanning tree's degrees if the constraints are also similar. In MDST algorithm, with the use of maximum degree heuristic, the diameter of the resulting spanning tree is decreased. According to the runtime results, we can say that self-stabilizing spanning tree algorithms are more sensible to the number of nodes than the classical approach. This difference is mainly caused by periodic updates of the nodes and self stabilization property. Although runtime of the self-stabilizing spanning tree algorithms increase when the size of the network grows, this increase is neither exponential nor linear. This sub-linear increase proves the scalability of the self-stabilizing algorithms as investigated in this study. The benefit is the self stabilization property which can drastically decrease the maintenance costs of spanning trees in large networks.

According to our experimental observations, we can conclude that in dynamic environments in which the nodes come and go, self stabilization algorithms can be used effectively if the runtime tradeoff is acceptable. Moreover, by using heuristics, resulting spanning trees can be tuned according to specific purposes.

References

[1] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *WDAG90: Proceedings of the 4th International Workshop on Distributed Algorithms*, pages 15–28. Springer-Verlag, 1991.

[2] M. Ahuja and Y. Zhu. A distributed algorithm for minimum weight spanning trees based on echo algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 2–8, 1989.

[3] G. Antonoiu and P. K Srimani. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers and Mathematics with Applications*, 30, 1995.

[4] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 230–240, 1987.

[5] H. Baala, O. Flauzac, J. Gaber, M. Bui, and T. El-Ghazawi. A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *Journal of Parallel and Distributed Computing*, 63(1):97–104, 2003.

[6] L. Blin, M. G. Potop-Butucaru, and S. Rovedakis. Self-stabilizing minimum-degree spanning tree within one from the optimal degree. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11. IEEE Computer Society, 2009.

[7] F. Butelle, C. Lavault, and M. Bui. A uniform self-stabilizing minimum diameter tree algorithm (extended abstract). In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 257–272. Springer-Verlag, 1995.

[8] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[9] S. Dolev. Optimal time self stabilization in dynamic systems (preliminary version). In *WDAG '93: Proceedings of the 7th International Workshop on Distributed Algorithms*, pages 160–173. Springer-Verlag, 1993.

[10] R.G. Gallagher, P.A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, 1983.

[11] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, 2003.

[12] S. K. Gupta and P. K. Srimani. Self-stabilizing multicast protocols for ad hoc networks. *Journal of Parallel and Distributed Computing*, 63(1):87–96, 2003.

[13] T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. Self-stabilizing spanning tree algorithm for large scale systems. 4280:574–575, 2006.

[14] N. Kotowski, A. A. B. Lima, E. Pacitti, P. Valduriez, and M. Mattoso. Parallel query processing for olap in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2039–2048, 2008.

[15] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[16] Y.N. Lien. A new node-join-tree distributed algorithm for minimum weight spanning trees. In *Proceedings of the 8th International Conference on Distributed Computing System*, pages 334–240, 1988.

[17] R. C. Pan, J. Z. Wang, and L. R. Chow. A self-stabilizing distributed spanning tree construction algorithm with a distributed demon. *Tamsui Oxford Journal of Mathematical Sciences*, 15:23–32, 1999.