Average Execution Time Analysis of a Self-stabilizing Leader Election Algorithm

Juan Paulo Alvarado-Magaña and José Alberto Fernández-Zepeda

CICESE
Department of Computer Science
Km. 107 Carretera Tijuana-Ensenada
Ensenada, B.C. 22860, Mexico
{alvarado, fernan}@cicese.mx

Abstract

This paper deals with the self-stabilizing leader election algorithm of Xu and Srimani [10] that finds a leader in a tree graph. The worst case execution time for this algorithm is $O(N^4)$, where N is the number of nodes in the tree. We show that the average execution time for this algorithm, assuming two different scenarios, is much lower than $O(N^4)$. In the first scenario, the algorithm assumes a equiprobable daemon and it only privileges a single node at a time. The average execution time for this case is $O(N^2)$. For the second case, the algorithm can privilege multiple nodes at a time. We eliminate the daemon from this algorithm by making random choices to avoid interference between neighbor nodes. The execution time for this case is $O(N)$. We also show that for specific tree graphs, these results reduce even more.

1. Introduction

A self-stabilizing system has the ability to go from any arbitrary global state to a legitimate global state in a finite number of steps without any outside intervention. Self-stabilization is useful to design fault tolerant distributed systems for transient faults. Dijkstra [4] proposed the concept of self-stabilization in 1974. Since then, self-stabilization has received a lot of attention and researchers have published a number of papers in this area. General references on self-stabilization are [2, 9].

A fundamental problem in distributing computing is leader election. Roughly speaking, leader election is the problem of selecting any one element from a given set of candidates. This problem has many variations depending on the assumptions about the features of the candidates; one of the most difficult variations to solve is when each candidate is indistinguishable from the others and when the communication among candidates is restricted.

Researcher have design many self-stabilizing algorithms that solve the leader election problem for different topologies. We can mention the following. Huang [7] and Itkis *et al.* [8] designed algorithms for bidirectional prime size uniform rings. Gosh and Gupta [6] and Fich and Johnen [5] designed algorithms for unidirectional prime size uniform rings. Dolev *et al.* [3] solved the problem for a general graph. Antonoiu and Srimani [1] designed an algorithm for a tree graph in which only an internal node can be leader. They proved that the algorithm terminates in finite time, but they did not perform temporal complexity analysis. Later, Xu and Srimani [10] simplified and generalized the algorithm of [1]. They argue that their algorithm can choose any node of the tree as leader and the execution time for the worst-case scenario is $O(N^4)$.

The worst-case execution time of an algorithm is a very useful parameter to characterize an algorithm; however, sometimes the worst-case execution time is very far from the typical execution time. When this happens, programmers may choose and implement a wrong algorithm to solve some specific problem, especially when the typical execution time of the algorithm is unknown. This is the case for the leader election algorithm of Xu and Srimani [10], whose worst execution time is $O(N^4)$. In their paper, they conjecture that the typical execution time may be much lower $O(N^4)$.

In this paper, we analyze the average execution time of this algorithm on arbitrary trees of N nodes under two assumptions. The first assumption is when the algorithm allows a single node to change its internal variables at a time; for this case, the average execution time is $O(N^2)$.The second assumption is when the algorithm allows multiple nodes to change their variables simultaneously; for this case, the average execution time is $O(N)$. For both cases, these values decrease considerably when the algorithm runs on specific types of trees.

We organize the remaining sections of this paper as follows. Section 2 briefly describes the algorithm of Xu and Srimani [10]. Section 3 provides basic notation and definitions. Section 4 discusses the behavior of the two main stabilizing rules of the algorithm. Section 5 and 6 presents the analysis of the average execution time of the algorithm for two different scenarios. Finally, Section 7 presents some concluding remarks.

2. Xu and Srimani's Self-Stabilizing Leader Election Algorithm

Xu and Srimani [10] designed a self-stabilizing leader election algorithm for arbitrary trees. This section provides a brief description of the algorithm.

2.1 Features and assumptions

The main features and assumptions of this algorithm are the following:

1. The worst execution time of the algorithm is $O(N^4)$, where N is the number of nodes in the tree.
2. The algorithm works on anonymous trees, so the algorithm does not have access to the index of each vertex.
3. The algorithm can choose any arbitrary vertex (internal or leaf) of the tree as a leader.
4. The algorithm does not assume any particular order in the execution of stabilizing operations of the vertices.
5. To avoid conflicts, the algorithm uses a variation of the central daemon to decide the node or nodes that execute stabilizing operations.

2.2 How it works

Each node i stores an integer variable x_i and a pointer p_i. Pointer p_i can take any value from the set $\{N(i) \cup i\}$, where $N(i)$ refers to the set of neighbors of node i. Each node i has access to variable x_j if $j \in N(i)$. At any step of the algorithm, node i compares its variable x_i against variables x_j of its neighbors. The algorithm checks whether the results of these comparisons and the value of p_i satisfy the conditions of any of the three stabilizing rules (described in Section 2.3). When a node i satisfies the conditions of a rule, it can execute the stabilizing operations of that rule (provided the daemon privileges node i). The values of each variable x and p change continuously until no node satisfies any of the three rules. At this point, the algorithm terminates and there is exactly one node i such that $p_i = i$ and $x_i > x_j$ for all $j \in V - \{i\}$; node i is the global maximum and the leader. An interesting property of the algorithm is that if node i is the leader and j any other node in the tree, then $x_i - x_j$ is the number of edges between nodes i and j. At the end, we can view this tree as a rooted tree where the leader is the root and each node points to the direction of the leader.

2.3 Stabilizing Rules

Each rule in the algorithm consists of some conditions that a node needs to satisfy to execute its stabilizing operations, provided the demon privileges the node. These rules are the following:

Rule R_1**.** If node i has at least two neighbors, say j and k, such that $x_j \geq x_i$ and $x_k \geq x_i$, then node i executes the following operations:

$x_i := \max\limits_{j \in N(i)} \{x_j\} + 1$
$P_i := i$

Rule R_2**.** If node i has exactly one neighbor j, such that $x_j \geq x_i$ AND $(x_i \neq x_j - 1$ OR $P_i \neq j)$, then node i executes the following operations:

$x_i := x_j - 1$
$P_i := j$

Rule R_3**.** If $x_i > x_j$ for all $j \in N(i)$ AND if $Pi \neq i$, then node i executes the following operation.

$P_i := i$.

3. Basic definitions

Let $T = (V_T, E_T)$ be an undirected tree, where $|V_T| = N$. A *candidate node* is a node that satisfies the conditions of any of the three stabilizing rules. Let C_α denote the set of candidate nodes that satisfy the conditions of rule R_α, where $\alpha \in \{1, 2, 3\}$. Let C_0 denote the set of nodes that are not candidates. When the daemon chooses a candidate node to execute some stabilizing operations, then this node is a *privileged node*. The *exter-level* and *inter-level* are labels with positive integer values that we assign to each node in a tree, according to its position in the tree. These labels facilitate the explanation of the algorithm in the paper. The nodes with exter-level zero are the leaves of the tree. The nodes with exter-level m are those nodes that became leaves when we remove from the tree all the nodes with exter-level $0, 1, ..., m-1$. The node with greatest exter-level is the center of the tree. Figure 1a shows a tree with exter-level labels inside the nodes. (The concept of exter-level is equivalent to the definition of the sets S_i from the paper of Xu and

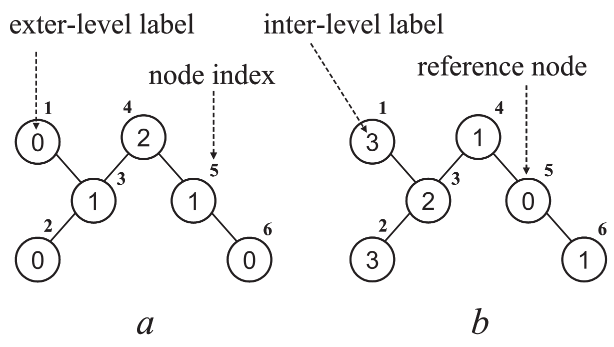

Figure 1. a) Tree with exter-level labels inside the nodes. b) Tree with inter-level labels inside the nodes; node 5 is the reference node

Srimani [10]; each node of the set S_m has exter-level m.). The inter-level of node j indicates the distance (number of edges) between node j and a specific reference node k. Figure 1b shows a tree with inter-level labels inside the nodes; notice that node 5 is the reference node.

The daemon is very useful abstraction that prevents a node to interfere with its neighbors. Since the goal of Xu and Srimani [10] was to calculate the worst execution time of the algorithm, they assumed an *adversary daemon*. This daemon, at each step of the algorithm, privileges the candidate that maximizes the execution time.

Our intention in this paper is to calculate the average execution time of the algorithm, so we assume an *equiprobable daemon* in the analysis of Section 5. This daemon, at each step of the algorithm and with the same probability, chooses only one candidate to execute its stabilizing operations. In Section 6, we remove the daemon from the algorithm.

Definition 1. Let $T = (V_T, E_T)$ be a tree and let k, $l \in V_T$ two arbitrary neighbor nodes. A sub-tree $T_{\neg(k,l)}$ of T is the tree that contains node k after removing edge (k, l) from T. □

Example 1: Notice that after removing edge $(5, 4)$ from the tree of Figure 2, the tree on the right is sub-tree $T_{\neg(5,4)}$ of T since it contains node 5.

Definition 2. A tree T (sub-tree $T_{\neg(k,l)}$) is stable with respect to (w.r.t.) rule R_1 if it satisfies the following conditions:

1. No node is candidate under rule R_1 in T ($T_{\neg k,l)}$).
2. There exist a node k that is a global maximum in T ($T_{\neg(k,l)}$), that is $x_k > x_i$, for all $i \neq k$. □

A consequence of the above conditions is the following. If a node j is located at a distance t from node k, the value of x_j is less than the values of all variables x of the nodes that are on the path between node j and node k.

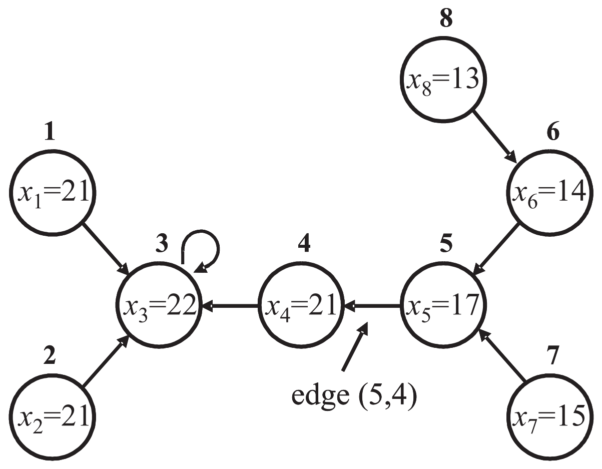

Figure 2. A tree that is stable w.r.t. rule R_1

Example 2: The tree of Figure 2 is stable w.r.t. rule R_1 and its global maximum is node 3. Notice, that sub-tree $T_{\neg(5,4)}$ is also stable w.r.t. rule R_1 and its global maximum is node 5.

Definition 3. Let $T = (V_T, E_T)$ be a tree. A sub-tree $T_{k(\delta)}$ of T is the tree generated after removing from T all the nodes with inter-level greater than δ, assuming that node k is the reference node. □

Example 3: Consider the tree of Figure 2. The sub-tree $T_{3(1)}$ is the tree integrated by nodes 1, 2, 3, and 4. Node $k = 3$ is the reference node and its inter-level label is zero. The inter-level labels for nodes 1, 2, and 4 are ones. The remaining nodes have inter-level labels greater than $\delta = 1$.

Definition 4. A tree $T = (V_T, E_T)$ (sub-tree $T_{k(\delta)}$ of T) is stable w.r.t. rule R_2 if it satisfies the following conditions:

1. Node $i \notin C_2$ for all i in the tree T (sub-tree $T_{k(\delta)}$).
2. There exists a node k that is a global maximum in the tree T (sub-tree $T_{k(\delta)}$).
3. For each node i in the tree T (sub-tree $T_{k(\delta)}$), $x_k - x_i$ is equal to the number of edges between nodes k and i. □

Example 4: Consider the tree of Figure 2. Notice that sub-tree $T_{3(1)}$ is stable w.r.t. rule R_2.

Since any node in a sub-tree stable w.r.t. rule R_1 (R_2) never executes rule R_1 (R_2) again, we can say that a node is stable w.r.t. rule R_1 (R_2) if it belongs to a sub-tree that is stable w.r.t. rule R_1 (R_2).

4. Global effect of rules R_1 and R_2

Let $T = (V_T, E_T)$ be a tree. To understand the behavior of the algorithm under rule R_1, assume that the algorithm only privileges nodes candidates under rule R_1. At the beginning, the stabilizing operations of rule R_1 can occur in any node of T (except the leaves); however, after some iterations, one can observe that these operations execute on nodes around a specific region of T. This region gradually reduces until, eventually, it becomes a single node, the global maximum. We can say that the stabilizing process of rule R_1 starts in the leaves of T and gradually propagates to the node that eventually becomes the global maximum. The position of the global maximum can be any at the end of the algorithm. This position depends on the initial values of variables x and the choices of the daemon.

To understand the behavior of the algorithm under rule R_2, assume that T is stable w.r.t. rule R_1. Assume that we assign inter-level labels to each node of the tree using node k, the global maximum, as reference node. Notice that node k is already stable w.r.t. rule R_2, because it belongs to subtree $T_{k(0)}$, which is stable w.r.t. rule R_2. Similarly, as the previous case, the stabilizing operations of rule R_2 can occur in any node (except the global maximum), but gradually these operations take place further from the global maximum. At the end, leaves are the last nodes that execute the operations. We can say that the stabilizing process w.r.t. rule R_2 starts at the global maximum and propagates towards the leaves of T.

5. Average execution time of the algorithm for the single-privileged node assumption

To facilitate the analysis, we split the algorithm in two phases. The first phase ends when $|C_1| = 0$ (*i.e.* when all the nodes in the tree are stable w.r.t. rule R_1). The second phase starts at this point and includes the remaining steps of the algorithm.

First, we compute the average number of steps the algorithm needs to finish phase one for an arbitrary tree.

5.1 Phase 1

Let y_m be the number of nodes in exter-level m. Our first goal is to calculate the average number of steps, T_m, necessary to privilege all the nodes in exter-level m. The following is a list with our assumptions.

1. A demon can privilege only one node at a time. The privileged node can be any from the set $\{C_1 \cup C_2 \cup C_3\}$ with the same probability. This is the *single-privileged node assumption*.
2. All the nodes with exter-level $< m$ are already stable w.r.t. rule R_1.
3. Each node in exter-level m is element of C_1
4. Each node not included in exter-level m is element of $\{C_2 \cup C_3\}$ and the daemon can privilege this node any number of times.

The following are some remarks.

1. Sets C_0, C_1, C_2, and C_3 are disjoint and $\{C_0 \cup C_1 \cup C_2 \cup C_3\} = V_T$.
2. In a typical scenario, nodes in exter-level m may belong to any of the sets C_0, C_1, C_2, or C_3, so, Assumption 3 is the worst-case scenario to stabilize nodes of exter-level m w.r.t. rule R_1.
3. Any stabilizing operation of rule R_2 or R_3 in a sub-tree stable w.r.t. rule R_1 keeps the sub-tree stable w.r.t. rule R_1.

Initially, the daemon can privilege a node of inter-level m with a probability y_m/N, since there are y_m candidates in exter-level m and N possible candidates in T. Let k_1 be a random variable that represents the number of steps required to privilege the first node of exter-level m. The expected value of k_1 is N/y_m, because k_1 has a geometric distribution. The average number of steps required to privilege the second candidate of inter-level m is $k_2 = N/(y_m - 1)$. In general, the average number of steps required to privilege the j-ary node of inter-level m is $k_j = N/(y_m - j + 1)$. Equation 1 computes the value of T_m.

$$T_m = \sum_{j=1}^{y_m} k_j = N \sum_{i=1}^{y_m} \frac{1}{i} = c_m N \log y_m, \qquad (1)$$

where c_m is a constant. Let $T_{sp-1}(N)$ be the average number of steps required to stabilize all the nodes of T w.r.t. rule R_1 ("*sp*" stands for single-privileged). If the maximum exter-level is r, then

$$\begin{aligned} T_{sp-1}(N) &= \sum_{m=1}^{r} T_m \leq cN(\log y_1 + \cdots + \log y_r) \\ &= O(N^2) \qquad (2) \end{aligned}$$

where $c = \max\{c_m\}$ for all m. Since $\sum_{\forall m} y_m = N$, then $\sum_{\forall m} \log y_m < N$. The values of y_m and r depend on the topology of the tree. (Notice that when the topology of the tree is a chain of nodes, $y_m = 2$ for all m, then $\sum_{\forall m} \log y_m = O(N)$.)

5.2 Phase 2

Now, our goal is to calculate the average number of steps, T_n, necessary to privilege all the nodes of inter-level n w.r.t. rule R_2. The following is a list with our assumptions.

1. Each node with inter-level $< n$ is already stable w.r.t. rule R_2.
2. Each node in inter-level n is candidate under rule R_2
3. Each node in any inter-level $> n$ is element of C_2 and the daemon can privilege this node any number of times.

Remark 1. We are not considering execution of rule R_3 in the analysis because the number of times the algorithm executes R_3 is much smaller than the number of times the algorithm executes rules R_1 and R_2.

Let y_n be the number of nodes in inter-level n and let z_n be the number of nodes in all the inter-levels equal or greater than n. The value of T_n is the following:

$$\begin{aligned} T_n &= \sum_{j=0}^{y_n-1} \frac{z_n - j}{y_n - j} = \sum_{i=1}^{y_n} \left(\frac{(z_n - y_n)}{i} \right) + y_n \\ &= c_n (z_n - y_n) \log y_n + y_n \end{aligned} \tag{3}$$

where c_n is a constant. Notice that $(z_n - y_n) = z_{n+1} < N$ for all n. Let $T_{sp-2}(N)$ be the average number of steps required to stabilize all the nodes of the tree w.r.t. rule R_2. If the maximum inter-level is s, then

$$\begin{aligned} T_{sp-2}(N) &= \sum_{n=1}^{s} T_n < \sum_{n=1}^{s} (cN \log y_n + y_n) \\ &= O(N^2) \end{aligned} \tag{4}$$

Thus, the average execution time of the algorithm with the single-privileged node assumption is:

$$T_{sp}(N) = T_{sp-1}(N) + T_{sp-2}(N) = O(N^2) \tag{5}$$

Section 7 shows that the average execution time is smaller than $O(N^2)$ for specific trees.

6. Average execution time for the multiple-privileged node assumption

In the previous analysis, we assume that the daemon privileges only a single node at each step. Now we allow the algorithm to privilege more than one candidate at a time. This is the *multiple- privileged node assumption*. For this purpose, we made the following changes:

1. At each step, each node i generates a random bit that stores in variable b_i.
2. We add extra conditions in the rules R_1 and R_2.
3. The execution of rule R_1 has higher priority than rule R_2.
4. We eliminate the daemon from the algorithm.

The modified rules are the following:

Rule R_{1+}. If node i has at least two neighbors, say j and k, such that $x_j \geq x_i$ and $x_k \geq x_i$ AND $b_i = 1$ AND $b_l = 0$ for all $l \in \{C_1 \cap N(i)\}$, then node i executes the following operations:

$x_i := \max_{j \in N(i)} \{x_j\} + 1$

$P_i := i$

Rule R_{2+}. If node i has exactly one neighbor j, such that $x_j \geq x_i$, AND $(x_i \neq x_j - 1$ OR $P_i \neq j)$ AND $\{N(i) \cap C_1\} = \emptyset$ AND $j \notin C_2$ then node i executes the following operations:

$x_i := x_j - 1$

$P_i := j$

Rule R_{3+}. It is the same as rule R_3.

The purpose of the additional conditions in rules R_{1+} and R_{2+} is to avoid two or more adjacent nodes to be privileged simultaneously. These changes give rule R_1 higher priority than rule R_2. With this modification, the execution time of the algorithm reduces considerably as we show.

Remark 2. A especial case occurs when there are no candidates under rule R_{1+} and there are two neighbor nodes, i and j, that are maximum. For this case, it is necessary that rule R_{2+} breaks the symmetry by generating random numbers b_i and b_j in each node and allowing one of them to execute the stabilizing operations of R_{2+}. Once the symmetry is broken, an additional operation of rule R_{1+} stabilizes the tree w.r.t. rule R_{1+}. This especial condition can be added to rule R_{2+}.

6.1 Phase 1

We follow the same procedure we use in Section 5.1. First, we calculate the average number of steps necessary to stabilize all the nodes of exter-level m w.r.t. rule R_{1+}. We assume that all the nodes with exter-level $< m$ are already stable w.r.t. rule R_{1+}.

Each node i of exter-level m may have a conflict with its neighbor j in exter-level $m+1$, if both are candidates under rule R_{1+}. (Notice that node i may have one or more neighbors in exter-level $m-1$ and only one neighbor in exter-level $m+1$. Since we assume that all nodes of exter-level $< m$ are already estable w.r.t. rule R_{1+}, the only possible conflict of node i is with its neighbor in exter-level $m+1$.) So only when $b_i = 1$ and $b_j = 0$ (this happens with probability $1/4$), node i privileges itself and executes the stabilizing operations of rule R_{1+}. If we assume the worst case, when all the y_m nodes of exter-level m are candidates under rule R_{1+}, on average, $y_m/4$ nodes privilege themselves after one step. This process continues and in each additional step, $1/4$ of the remaining candidates privilege themselves. To calculate the average number of steps required to stabilize all nodes, except one, of exter-level m w.r.t. rule R_{1+}, we solve Equation 6.

$$\left(\frac{3}{4}\right)^{\alpha} y_m = 1 \implies \alpha = c_m \log y_m \tag{6}$$

where c_m is a constant. Additionally, on average, the algorithm needs four steps to privilege the last candidate of exter-level m. Let T_m be the average number of steps required to stabilize all nodes of exter-level m w.r.t. rule R_{1+}. Equation 7 gives the value of T_m.

$$T_m = c_m \log y_m + O(1) \tag{7}$$

Let $T_{mp-1}(N)$ be the average number of steps required to stabilize all the nodes of the tree w.r.t. rule R_{1+} ("*mp*" stands for multiple-privileged). Assume that the maximum exter-level is r.

$$T_{mp-1}(N) = \sum_{m-1}^{r} T_m = O(N) \tag{8}$$

6.2 Phase 2

Now, our goal is to calculate the average number of steps, T_n, necessary to privilege all the nodes of inter-level n w.r.t. rule R_2. Our assumptions for this phase are the following.

1. All the nodes with inter-level $< n$ are already stable w.r.t. rule R_2+.
2. All nodes in inter-level n are candidate under rule R_2+
3. Any node in any inter-level $> n$ belongs to $\{C_2\}$ and they can privilege any number of times.

Since all the nodes in inter-level $(n-1) \notin \{C_2\}$ and all nodes in the tree are stable w.r.t. rule R_1+, then $T_n = O(1)$. Let $T_{mp-2}(N)$ be the average number of steps required to stabilize all the nodes of the tree w.r.t. rule R_{2+}. Assume that maximum inter-level is s (In the worst case the height s of the tree is $O(N)$).

$$T_{mp-2}(N) = \sum_{n-1}^{s} T_n = O(s) = O(N) \tag{9}$$

Thus, the average execution time of the algorithm with the multiple-privileged node assumption is:

$$T_{mp}(N) = T_{mp-1}(N) + T_{mp-2}(N) = O(N) \tag{10}$$

Notice that the average execution time is smaller than $O(N)$ for specific trees, as shown in Section 7.

7. Analysis for specific tree topologies

The average execution time of the algorithm for the single-privileged node assumption is $O(N^2)$, for an arbitrary tree of N nodes. Notice that by replacing the values of y_m, y_n, z_n, r and s, in Equations 1 to 4, for the parameters of specific types of trees, the value of $T_{sp}(N)$ reduces greatly, as shown in Table 1.

The average execution time of the algorithm for the multiple-privileged node assumption is $O(N)$, for an arbitrary tree of N nodes. Table 2 shows that this time reduces for specific types of trees. Compute $T_{mp}(N)$ by replacing the values of y_m, y_n, z_n, r and s in Equations 6 to 9.

Table 1. Average execution time of the algorithm of Xu and Srimani [10] for different types of trees for the single-privileged node assumption.

Type of tree (N nodes)	Average execution time, $T_{sp}(N)$
Arbitrary tree	$O(N^2)$
Tree with diameter $O(\log N)$	$O(N \log^2 N)$
Double depth logarithmic tree	$O(N \log N \log\log N)$

Table 2. Average execution time of the algorithm of Xu and Srimani [10] for different types of trees for the multiple-privileged node assumption.

Type of tree (N nodes)	Average execution time, $T_{sp}(N)$
Arbitrary tree	$O(N)$
Tree with diameter $O(\log N)$	$O(\log^2 N)$
Double depth logarithmic tree	$O(\log N \log\log N)$

8. Concluding remarks

We compute an upper bound on the average execution time of the self-stabilizing algorithm for leader election of Xu and Srimani [10]. We present the analysis for two different scenarios. The first scenario assumes that the daemon privileges only one node at time, the second assumes that the algorithm privileges more than one node at a time. The average execution times for these scenarios are $O(N^2)$ and $O(N)$, respectively. In the second scenario, we made some minor changes to the rules to avoid two or more adjacent nodes to be privileged at the same time. With these changes, the daemon is not required. We also show that for some common tree topologies, the algorithm has a better performance. These results show that the typical execution time of the algorithm is much smaller than $O(N^4)$, the worst case scenario.

References

[1] G. Antonoiu and P. Srimani, A Self-Stabilizing Leader Election Algorithm for Tree Graphs. *Journal of Parallel and Dis-*

tributed Computing, 34(2):227–232, 1996.

[2] S. Dolev. Self-Stabilization. The MIT press, Cambridge Massachusetts, 2000.

[3] S. Dolev, A. Israeli, and S. Moran. Uniform Dynamic Self-Stabilizing Leader Election. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):424–440, 1997.

[4] E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[5] F. Fich and C. Johnen. A Space Optimal, Deterministic, Self-Stabilizing, Leader Election Algorithm for Unidirectional Rings. *In proc. of the 15th International Symposium on Distributed Computing*, pages 224–239, 2001.

[6] S. Ghosh and A. Gupta. An Exercise in Fault-Containment: Self-Stabilizing Leader Election. *Information Processing Letters*, 59(5):281–288, 1996.

[7] S. Huang. Leader Election in Uniform Rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, 1993.

[8] G. Itkis, C. Lin, and J. Simon. Deterministic, Constant Space, Self-Stabilizing Leader Election on Uniform Rings. *In Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 288–302, 1995.

[9] M. Schneider. Self-Stabilization. *ACM Computing Surveys* 25(1):45–67, 1993.

[10] Z. Xu and P. K. Srimani. Self-Stabilizing Anonymous Leader Election in a Tree. *International Journal of Foundations of Computer Science*, 17(2):323–335, 2006.