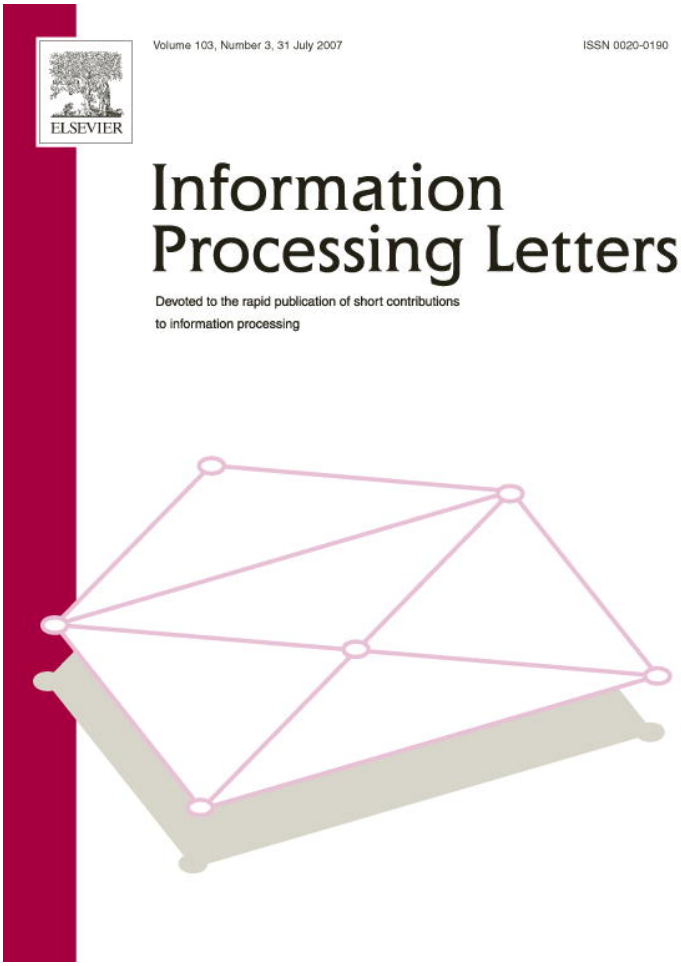

Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 103 (2007) 88–93
Information Processing Letters
www.elsevier.com/locate/ipl

Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler

Volker Turau

Hamburg University of Technology, Institute of Telematics, Schwarzenbergstraße 95, 21073 Hamburg, Germany

Received 21 July 2006; received in revised form 19 February 2007

Available online 12 March 2007

Communicated by A.A. Bertossi

Abstract

This paper presents distributed self-stabilizing algorithms for the maximal independent and the minimal dominating set problems. Using an unfair distributed scheduler the algorithms stabilizes in at most $\max\{3n-5, 2n\}$ resp. $9n$ moves. All previously known algorithms required $O(n^2)$ moves.

Keywords: Self-stabilizing algorithms; Fault tolerance; Distributed computing; Graph algorithms

1. Introduction

The concept of self-stabilization—introduced by Dijkstra [3]—is considered to be a very general technique to design a system to tolerate arbitrary transient faults. A distributed system is self-stabilizing if it can start at any possible global configuration and regain consistency in a finite number of steps by itself without any external intervention and remains in a consistent state [4]. The state of a fault-free system is defined by a local predicate based on the states of the nodes. The paradigm has been developed for different communication styles. In the shared-variable version, every node executes the same program, and maintains and changes its own state based on its current state and the states of its neighbors. In wireless systems nodes broadcast their state to their neighbors after every change of the state. The program of every node consists of a set of rules of the form:

$$\langle \text{precondition} \rangle \rightarrow \langle \text{statement} \rangle.$$

The precondition of a rule is a boolean expression involving the state of the node itself and its neighbors. The statement updates the state of the node only. The execution of a statement is called a move. It is assumed that rules are atomically executed, i.e., the evaluation of a precondition and the move are performed in one atomic step. A rule is said to be enabled if its precondition evaluates to true. A node is enabled if at least one of its rules is enabled. Self-stabilizing systems operate in rounds. In every round, first, all nodes check the preconditions of their rules. Then a scheduler selects a subset of the enabled nodes to make a move. Common schedulers are the central scheduler (only a single node makes a move in every round), the unfair distributed scheduler (any nonempty subset of the enabled nodes can make their moves simultaneously), and the fully distributed or syn-

E-mail address: turau@tuhh.de.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2007.02.013

chronous scheduler (all enabled nodes make their moves simultaneously). Although it is easier to prove stabilization for algorithms working under the central scheduler, the fully distributed and the unfair distributed scheduler are more suitable for practical implementations. Note that the unfair distributed scheduler subsumes the other two types of schedulers and is the most general concept.

The construction of maximal independent sets (MIS) and minimal dominating sets (MDS) in distributed systems has attracted a lot of research due to the importance of the concept for many applications, e.g., clustering in wireless networks [1]. Self-stabilizing algorithms for these two problems based on the central scheduler making at most $2n$ (resp., $(2n+1)n$) moves have been proposed by various authors [6,8]. To make use of these algorithms with a distributed scheduler two transformation techniques are available: randomization [9] and local mutual exclusion [2]. In the first case the algorithms are only probabilistically self-stabilizing and in the second case unique process identifiers are required. Usually the transformed algorithms do not stabilize as fast as problem-specific algorithms. Two very similar algorithms for the MIS problem based on the fully distributed scheduler requiring unique node identifiers have been developed [5,7]. Both algorithms stabilize in $O(n)$ rounds, but require $O(n^2)$ moves in the worst case. Xu et al. present an algorithm for the MDS problem using a distributed scheduler that stabilizes after at most $4n$ rounds [10]. An analysis with respect to the required number of moves is not given. If an algorithm stabilizes after at most m moves (resp., r rounds) using an unfair distributed scheduler, then these bounds are valid for the fully distributed scheduler, but note that the opposite is not necessarily true.

In wireless systems with bounded resources the number of moves is at least as important as the number of rounds. The reason is that nodes broadcast their state after every move to their neighbors. Since communication is the main consumer of energy, a reduction of the number of messages prolongs the lifetime of a network.

The MIS and MDS algorithms presented in this work are the first algorithms to our knowledge that require only $O(n)$ moves in the worst case using an unfair scheduler. The algorithms require unique node identifiers.

2. Maximal independent sets

Let $G = (V, E)$ be a connected undirected graph. An independent set (IS) S of G is a subset of V such that $(u, v) \notin E \ \forall u, v \in S$. S is a maximal independent set (MIS) if any node v not in S has a neighbor in S. The self-stabilizing algorithms for the MIS problem in [6,8] are based on two simple rules that maintain a set I:

1. A node having no neighbor in I joins I.
2. A node in I having a neighbor in I leaves I.

To make this algorithm stabilize under a distributed scheduler it must be avoided that neighboring nodes simultaneously join or leave I but still being enabled in the following round. The idea of Ikeda et al. is to reduce the number of neighboring nodes executing rule 2 in the same round [7]. This is achieved by allowing nodes to leave I only in case they have a neighbor in I that has a lower identifier. Thus, among the nodes where rule 2 is enabled, at least the node with the smallest identifier does not execute rule 2 in this round. They proved that this algorithm stabilizes after at most $(n+2)(n+1)/4$ moves using the unfair scheduler. Goddard et al. changed both rules: a node joins (resp., leaves) I if it does not have a neighbor in I with higher identifier (resp., if it has a neighbor in I with higher identifier) [5]. This did not break the $O(n^2)$ limit of necessary moves. They proved that using the fully distributed scheduler, the algorithm stabilizes in n rounds, but still makes $O(n^2)$ moves in the worst case (e.g., for a simple path with n nodes). The authors of [5] do not consider an unfair scheduler, but their result about the rounds is not true for the unfair scheduler. A big disadvantage of this algorithm is that even if the set I is a MIS, some nodes may still be enabled. The reason is that the algorithm produces a MIS that favors nodes with higher identifiers.

To avoid situations where neighboring nodes execute rules simultaneously, nodes need to know whether one of their neighbors is enabled. The above cited algorithms do this for nodes where rule 2 is enabled. They make use of the fact that if for a node v rule 2 is enabled, then there exists at least one neighbor of v where rule 2 is also enabled. But in general a node cannot tell from its own state and the state of a neighboring node alone whether a neighbor is enabled, in particular this is true for rule 1: the fact that a node has no neighbor in I does not allow to make this assumption about any of its neighbors. A solution for this problem presented in this paper is to use an additional state which indicates that a node is ready for making a move with rule 1. This reduces the worst case number of moves to $O(n)$.

The proposed algorithm uses three states. The state is defined by the variable *state*, its range of values is: IN, OUT and WAIT. The value IN means that the node is part of the MIS and OUT indicates that the node is not part of the MIS. The state WAIT signals that a node

wants to change into state IN, it may do so provided it has no neighbor with the same state but lower identifier. To formally define the rules the following predicates defined for each node v are needed:

- $inNeighbor(v) \equiv \exists w \in N(v)$: $w.state = \text{IN}$.
- $waitNeighborWithLowerId(v) \equiv \exists w \in N(v)$: $w.state = \text{WAIT} \wedge w.id < v.id$.
- $inNeighborWithLowerId(v) \equiv \exists w \in N(v)$: $w.state = \text{IN} \wedge w.id < v.id$.

The self-stabilizing algorithm $\mathcal{A}_{\text{MIS}}$ uses the following four rules:

1. $state = \text{OUT} \wedge \neg inNeighbor(v) \rightarrow state := \text{WAIT}$.
2. $state = \text{WAIT} \wedge inNeighbor(v) \rightarrow state := \text{OUT}$.
3. $state = \text{WAIT} \wedge \neg inNeighbor(v) \wedge \neg waitNeighborWithLowerId(v) \rightarrow state := \text{IN}$.
4. $state = \text{IN} \wedge inNeighbor(v) \rightarrow state := \text{OUT}$.

Lemma 2.1. *In any configuration in which no node is enabled the set $I = \{v \mid v.state = \text{IN}\}$ is a maximal independent set of G.*

Proof. Suppose there exists a node with state WAIT. Let v be such a node with minimal id. Since the precondition of rule 2 is not satisfied, v has no neighbor with state IN. Since v has no neighbor w with state WAIT and $w.id < v.id$, rule 3 is enabled. This contradiction shows that there is no node with state WAIT. Since rule 4 is not enabled, the set I is independent. Finally, I cannot be extended because rule 1 is not enabled. □

Let I be any maximal independent set for G and let the state of each node in I be IN and OUT for nodes outside I. Then no rule is enabled. Thus, if the algorithm is initialized with a MIS, then this set is not changed again.

Lemma 2.2. *If a node executes rule* 3 *then it will never again execute a rule and each neighbor of this node will execute at most one more rule and this will be rule* 2*.*

Proof. Let v be a node that executes rule 3. At this instant all neighbors of v have state OUT or WAIT and those with state WAIT have a higher identifier than v. Thus, none of the neighbors of v can execute rules 3 and 4 in this round because the preconditions of these rules are not satisfied. Hence, the neighbors of v can only execute rules 1 or 2 in the same round. This implies that after this round v has state IN and all neighbors of v have state OUT or WAIT. The only rule that v can execute next is rule 4, but in order to do so, one of its neighbors would have to change into state IN with rule 3. But as long as v is in state IN this is impossible. Therefore, v will never execute a rule again. Furthermore, only neighbors of v with state WAIT can execute a rule and this is rule 2. After that execution they have state OUT and cannot execute another rule. □

Lemma 2.3. *Only the following four sequences of states and their suffixes are possible for each node during the execution of algorithm $\mathcal{A}_{\text{MIS}}$ using an unfair scheduler*:

WAIT OUT WAIT OUT

WAIT OUT WAIT IN

IN OUT WAIT OUT

IN OUT WAIT IN

IN OUT

Proof. (a) Consider a node v initially in state WAIT. In this state v can only execute rules 2 and 3. When v executes rule 3 then according to Lemma 2.2 v changes into state IN and remains in that state forever. This corresponds to the sequence WAIT IN. Otherwise v executes rule 2 and moves to state OUT. Consider the case that v executes another rule. This can only be rule 1 and thus v cannot have a neighbor in state IN at that instant. Now v is in state WAIT. In case no neighbor of v changes to state IN before v executes again, this execution must be with rule 3. According to Lemma 2.2 the node never executes a rule again. This corresponds to the sequence WAIT OUT WAIT IN. Otherwise a neighbor of v has in the mean time changed to state IN using rule 3. Then node v only makes one more move using rule 2. This corresponds to the sequence WAIT OUT WAIT OUT.

(b) Consider a node v initially in state OUT. v can only execute rule 1, bringing the node into state WAIT. Note, that at this moment, v has no neighbor with state IN. If before the next move of v no neighbor has changed its state to IN, v will execute rule 3 and then it will never make a move again. This leads to the sequence OUT WAIT IN. Otherwise a neighbor of v changes to state IN and as before, v makes only one more move giving rise to sequence OUT WAIT OUT.

(c) Consider a node v initially in state IN. Then v can only execute rule 4 bringing the node into state OUT. In order to execute again all neighbors must change their state to OUT or WAIT. Then v can execute rule 1 leading to state WAIT. As before, the node can now execute only rule 2 or rule 3. This leads to one of the sequences IN OUT WAIT IN or IN OUT WAIT OUT. □

The following theorem follows immediately from the last three lemmata.

Theorem 2.1. *Algorithm $\mathcal{A}_{\text{MIS}}$ is self-stabilizing under an unfair distributed scheduler and stabilizes after at most $3n$ moves with a maximal independent set, where n is the number of nodes. This bound is attained.*

To see that this bound is really attained, consider n nodes arranged with ascending identifiers along a line, were each node is connected to its neighbors. If initially all nodes are in state IN, then it is possible that all nodes first change into state OUT and then into state WAIT. Finally all nodes will then change into their stable state, where the nodes on the line are alternating in state IN and OUT. Thus, each node makes 3 moves.

The total number of moves can be further reduced by slightly changing rule 4 as follows:

$4'$. $state = \text{IN} \wedge inNeighborWithLowerId(v) \rightarrow state := \text{OUT}$.

Corollary 2.1. *With rule $4'$ algorithm $\mathcal{A}_{\text{MIS}}$ is self-stabilizing under an unfair distributed scheduler and stabilizes in at most $\max\{3n-5, 2n\}$ moves. This bound is attained.*

Proof. Clearly the statements of Lemmas 2.1–2.3 still hold. First, consider the case that initially no node is in state IN. By Lemma 2.2 rule 4 is never executed and if a node executes rule 2, it will never execute a rule again. Thus by Lemma 2.3, every node makes at most 2 moves, in total at most $2n$ moves. Now consider the case that initially there is at least one node with state IN. Then the node in state IN with the smallest identifier never makes a move and its neighbors make at most one move. Since each node has at least one neighbor, the bound follows directly from Lemma 2.3.

To see that this bound is really attained, consider again $n > 4$ nodes arranged with ascending identifiers along a line, where each node is connected to its neighbors. If initially all nodes are in state IN and the scheduler selects the nodes repeatedly from right to left, then $3n-5$ moves are made before stabilization. □

3. Minimal dominating sets

A dominating set (DS) S of G is a subset of V such that each $v \in V \backslash S$ has at least one neighbor in S. S is a minimal dominating set (MDS) if for any node $v \in S$ the set $S \backslash \{v\}$ is not dominating. A maximal independent set is also a minimal dominating set, but the opposite is in general not true. Since it is desirable that a self-stabilizing algorithm initialized with a minimal dominating set does not make any move, MIS-algorithms are no suitable solution. The self-stabilizing algorithm for the MDS problem in [10] stabilizes after at most $4n$ rounds using the fully distributed scheduler.

Our algorithm for the MDS problem is an extension of the algorithm from the previous section. Apart from the variable *state*, each node has an additional variable *dependent* that points to another node. In case a node v has a single neighbor w with state IN then $v.dependent = w$ and if it has more than one neighbor with state IN or has itself state IN then $v.dependent = \Lambda$. The following additional predicates defined for each node v are needed:

- $uniqueInNeighbor(w, v) \equiv \exists\, unique\, w \in N(v)$: $w.state = \text{IN}$.
- $dependentNeighbors(v) \equiv \exists\, w \in N(v)$: $w.dependent = v$.

The self-stabilizing algorithm $\mathcal{A}_{\text{MDS}}$ uses the following seven rules:

1. $state = \text{OUT} \wedge \neg\, inNeighbor(v) \rightarrow state := \text{WAIT}$.
2. $state = \text{WAIT} \wedge inNeighbor(v) \rightarrow state := \text{OUT}$.
3. $state = \text{WAIT} \wedge \neg\, inNeighbor(v) \wedge \neg\, waitNeighborWithLowerId(v) \rightarrow state := \text{IN}$, $dependent := \Lambda$.
4. $state = \text{IN} \wedge inNeighbor(v) \wedge \neg\, dependentNeighbors(v) \rightarrow state := \text{OUT}$.
5. $state = \text{IN} \wedge \neg\, dependent = \Lambda \rightarrow dependent := \Lambda$.
6. $state = \text{OUT} \wedge uniqueInNeighbor(w, v) \wedge \neg\, dependent = w \rightarrow dependent := w$.
7. $state = \text{OUT} \wedge moreThanOneInNeighbor(v) \wedge \neg\, dependent = \Lambda \rightarrow dependent := \Lambda$.

Lemma 3.1. *In any configuration in which no node is enabled there is no node with state* WAIT *and the set $D = \{v \mid v.state = \text{IN}\}$ is a minimal dominating set for G.*

Proof. Suppose there exists a node with state WAIT. Let v be such a node with minimal *id*. Since the precondition of rule 2 is not satisfied, v has no neighbor with state IN. Since v has no neighbor w with state WAIT and $w.id < v.id$, rule 3 is enabled. This contradiction shows that there is no node with state WAIT.

Since rule 1 is not enabled, all nodes with state OUT have a neighbor in D, hence D is dominating. Suppose there exists a node $v \in D$ such that $D \backslash \{v\}$ is dominating. Hence, v has state IN and there exists $w_1 \in$

$N(v) \cap D$. By rule 4 there exists $w_2 \in N(v)$ such that $w_2.dependent = v$ and because of rule 5 node w_2 has state OUT. Then $w_2 \notin N(w_1)$ since rule 7 is not enabled for node w_2. Since $D\backslash\{v\}$ is dominating, there exists $w_3 \in N(w_2) \cap D\backslash\{v, w_1\}$. This yields $w_2.dependent = \Lambda$ since rule 7 is not enabled for node w_2. This contradiction proves that D is a minimal dominating set for G. □

Let D be any minimal dominating set for G and let the state of each node in D be IN and OUT for nodes outside D. Furthermore, set the value of the pointer *dependent* of each node according to the rules 5–7. Then no rule is enabled.

Lemma 3.2. *If a node executes rule* 3 *then it will never again execute a rule.*

Proof. Let v be a node that executes rule 3. At this instant all neighbors of v have state OUT or WAIT and those with state WAIT have a higher identifier than v. Thus, none of the neighbors of v can execute rules 3, 4, or 5 in this round because the preconditions of these rules are not satisfied. Hence, the neighbors of v can only execute rules 1, 2, 6, and 7 in the same round. This implies that after this round v has state IN and all neighbors of v have state OUT or WAIT. The only rule that v can execute next is rule 4, but in order to do so, one of its neighbors would have to change into state IN with rule 3. But as long as v is in state IN this is impossible. Therefore, v will never execute a rule again. □

Lemma 3.3. *Let w be a neighbor of a node v that just has executed rule* 3*. Then w will at most make* 4 *moves before stabilization.*

Proof. The node w is in state OUT or WAIT. If the state is WAIT, then w can only execute rule 2 and change to state OUT. If w is in state OUT it can only execute rules 6 and 7 until stabilization, thus the node remains forever in state OUT. In order to execute again after the first execution of rule 6 or 7, the number of neighbors of w with state IN has to change. Since v remains in state IN forever, this number is at least 1. If another neighbor of w changes into state IN (by executing rule 3), then w can execute at most rule 7 and then w will never execute again. The only other possibility is that the number of neighbors of w in state IN decreases. Only in case this number drops from 2 to 1, rule 6 is executed. But this can happen only once. Hence, the longest sequence of rule executions for w is: 2, 7, 6, 7. This proves the lemma. □

Lemma 3.4. *During the execution of algorithm $\mathcal{A}_{\mathrm{MDS}}$ each node makes at most* 9 *moves.*

Proof. (a) Consider a node v initially in state OUT, v can execute rules 1, 6 and 7. (i) Suppose that at the first execution of node v it has no neighbor in state IN. Then only rule 1 is enabled, moving the node into state WAIT. Consider the next execution of node v. In case the node still does not have neighbor with state IN, then it will execute rule 3 and will not execute again. On the other hand, if v has a neighbor with state IN, then this node has executed rule 3 in the mean time. By Lemma 3.3, v will make at most four more moves, leading to a total of 5 moves for this case. (ii) Suppose that at the first execution of node v it has at least one neighbor in state IN. Then at most one of the rules 6 and 7 can be enabled, both leaving the node in state OUT. After the execution of one of these rules only the value of the variable *dependent* is changed. In case at the next move of v, the node has an additional neighbor in state IN, a neighbor of v has executed rule 3 in the mean time and Lemmas 3.2 and 3.3 show that the node makes at most 5 moves. Otherwise, the number of neighbors in state IN has decreased. Because v made a move, this number has either dropped from 2 to 1 (rule 6 was executed) or from 1 to 0 (rule 1 was executed). In the first case, after the execution of the rule either v loses its remaining neighbor with state IN enabling rule 1 and after this move the node makes at most four more moves as shown above or it receives a new neighbor with state IN also leading to at most four more moves. Thus, in total at most 7 moves. The case in which rule 1 was executed by node v can treated similarly and leads to at most 6 moves. Summarizing, a node initially in state OUT makes at most 7 moves.

(b) Consider a node v initially in state WAIT. In this state v can only execute rules 2 and 3. By Lemma 3.2 it is only necessary to consider the case that v executes rule 2, i.e., v has a neighbor in state IN. Then the node is in state OUT. From part (a) of this proof it is known that v makes at most 7 more moves. This leads to a total of 8 moves for this case.

(c) Consider a node v initially in state IN, v can execute rules 4 and 5. In case rule 5 is executed first, the node is still in state IN and now can only execute rule 4 bringing the node into state OUT. From part (b) of this proof it is known that v makes at most 8 more moves. This leads to a total of 9 moves for this case. □

The following theorem follows immediately from the last four lemmata.

Table 1
Number of moves before stabilization of the two variants of each algorithm

Average node degree	3.8	6.5	14.3
$\mathcal{A}_{\text{MIS}}$	856	1063	1286
Modified $\mathcal{A}_{\text{MIS}}$	413	435	489
$\mathcal{A}_{\text{MDS}}$	1578	1817	1891
Modified $\mathcal{A}_{\text{MDS}}$	968	1053	1212

Theorem 3.1. *Algorithm $\mathcal{A}_{\text{MDS}}$ is self-stabilizing under an unfair distributed scheduler and stabilizes after at most $9n$ moves with a minimal dominating set, where n is the number of nodes.*

As in the previous section, rule 4 can also be changed by replacing the predicate *inNeighbor* with *inNeighborWithLowerId*.

4. Conclusion

This paper presented the first distributed self-stabilizing algorithms for the maximal independent and for the minimal dominant set problem that require a linear number of moves using an unfair distributed scheduler. All previously known algorithms required $O(n^2)$ moves.

Simulations indicate that the modified algorithms need far less moves on the average than $\mathcal{A}_{\text{MIS}}$ (resp., $\mathcal{A}_{\text{MDS}}$). Table 1 lists some results of simulations of the algorithms for three classes of unit disk graphs with 500 nodes using the fully distributed scheduler. This class of graphs is important for the modeling of wireless communication in general. The algorithms were applied to 5000 randomly generated unit disk graphs with the average node degree given in the table. The table lists the average number of moves before stabilization, the algorithms needed about 5 to 9 rounds before stabilization. Simulations with other graph classes and other sizes yielded similar results.

It is not known whether the number of moves can be reduced to $2n$. The determination of the expected number of moves of the algorithms is another open problem.

References

[1] K. Alzoubi, P.J. Wan, O. Frieder, Maximal independent set, weakly-connected dominating set, and induced spanners in wireless ad hoc networks, International Journal of Foundations of Computer Science 14 (2) (2003) 287–303.
[2] J. Beauquier, A.K. Datta, M. Gradinariu, F. Magniette, Self-stabilizing local mutual exclusion and daemon refinement, Chicago Journal of Theoretical Computer Science 1 (2002).
[3] E. Dijkstra, Self stabilizing systems in spite of distributed control, Comm. ACM 17 (11) (1974) 643–644.
[4] S. Dolev, Self-Stabilization, MIT Press, 2000.
[5] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks, in: Proc. Int. Parallel and Distributed Processing Symposium (2003).
[6] S.M. Hedetniemi, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Self-stabilizing algorithms for minimal dominating sets and maximal independent sets, Computer Mathematics and Applications 46 (5–6) (2003) 805–811.
[7] M. Ikeda, S. Kamei, H. Kakugawa, A Space-optimal self-stabilizing algorithm for the maximal independent set problem, in: Proc. 3rd Int. Conf. on Parallel and Distributed Computing, Applications and Technologies, 2002, pp. 70–74.
[8] S.K. Shukla, D.J. Rosenkrantz, S.S. Ravi, Observations on self-stabilizing graph algorithms for anonymous networks, in: Proc. 2nd Workshop on Self-Stabilizing Systems, 1995, pp. 7.1–7.15.
[9] V. Turau, C. Weyer, Randomized self-stabilizing algorithms for wireless sensor networks, in: Proc. Int. Workshop on Self-Organizing Systems, 2006, pp. 74–89.
[10] Z. Xu, S.T. Hedetniemi, W. Goddard, P.K. Srimani, A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph, in: Proc. 5th Int. Workshop on Distributed Computing, 2003, pp. 26–32.