

# Self-stabilizing Algorithms for Minimal Dominating Sets and Maximal Independent Sets

*S.M. Hedetniemi, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani*

Department of Computer Science

Clemson University

Clemson, SC 29634-0974 USA

{shedet, hedet, dpj, srimani}@cs.clemson.edu

May 15, 2001

## Abstract

In the self-stabilizing algorithmic paradigm for distributed computation each node has only a local view of the system, yet in a finite amount of time, the system converges to a global state satisfying some desired property. In this paper we present polynomial time self-stabilizing algorithms for finding a dominating bipartition, a maximal independent set, and a minimal dominating set in any graph.

**Keywords:** graph, self-stabilizing algorithm, dominating set, independent set.

## 1 Introduction

A distributed system can be modeled with an undirected graph  $G = (V, E)$ , where  $V$  is a set of  $n$  nodes and  $E$  is a set of  $m$  edges. If  $i$  is a node, then  $N(i)$ , its *open neighborhood*, denotes the set of nodes to which  $i$  is adjacent, and  $N[i] = N(i) \cup \{i\}$  denotes its *closed neighborhood*. Every node  $j \in N(i)$  is called a *neighbor* of node  $i$ . Throughout this paper we assume  $G$  is connected and  $n > 1$ .

*Self-stabilization* is a paradigm for distributed systems that allows the system to achieve a desired global state, even in the presence of faults [6, 7]. A fundamental idea of self-stabilizing algorithms is that no matter what global state the system finds itself in, after a finite amount of time the system will reach a correct and desired global state. Although the concept of self-stabilization was introduced in 1974 by Dijkstra [6], serious work on self-stabilizing algorithms did not start until the late 1980's. In a self-stabilizing algorithm, each node maintains its local variables, and can make decisions based only on its local variables and the contents of its neighbor's local variables. The contents of a node's local variables constitute its *local state*.

The system's *global state* is the union of all local states.

A node  $i$  may change its local state by making a *move*, i.e., changing the value of at least one of its local variables. Self-stabilizing algorithms are often given as a set of rules of the form  $p(i) \Rightarrow M$ , where  $p(i)$  is a predicate and  $M$  is a move. The predicate  $p(i)$  is defined in terms of the local state of  $i$  and the local states of its neighbors  $j \in N(i)$ . A node  $i$  becomes *privileged* if at least one predicate  $p(i)$  is true. When a node becomes privileged, it may (or may not) execute the corresponding move. We say the system has *stabilized* if no nodes are privileged.

We assume a serial model in which no two nodes move simultaneously. A central daemon selects, among all privileged nodes, the next node to move. If two or more nodes are privileged, we cannot predict which node will move next. An execution of self-stabilizing algorithm is represented by a sequence of moves  $M_1, M_2, \dots$ , in which  $M_s$  denotes the  $s$ -th move. The system's initial state is denoted by  $s_0$ , and for  $t > 0$ , the state resulting from  $M_t$  is denoted by  $s_t$ .

In this paper we focus on the design and analysis of self-stabilizing graph algorithms. That is, we seek self-stabilizing algorithms for identifying sets of nodes or sets of edges which satisfy a given property  $\mathcal{P}$ . Previous work in this area has produced self-stabilizing algorithms for centers and medians of trees [2, 3]; maximal matchings [12, 15, 11] graph colourings [8, 14]; shortest paths [16, 4]; articulation points [5]; and spanning trees [1, 13]. In many of these papers, correctness proofs for algorithms are given, but an analysis is not provided.

Recall that  $S \subseteq V$  is a *dominating* set if  $N(i) \cap S \neq \emptyset$  for every  $i \in V - S$ . Said another way, every node in  $V$  is either a member of  $S$  or is a neighbor of some member of  $S$ . A dominating set  $S$  is *minimal dominating* if no proper subset is dominating. Many papers have been written on graph domination and its generalizations. See [9, 10] for a comprehensive treatment of this large body of work. A *dominating bipartition* is a partition of  $V$  into two disjoint dominating sets. A set  $S \subseteq V$  is *independent* if no two members of  $S$  are adjacent. An independent set is *maximal independent* if no independent set properly contains it. In this paper we present linear and polynomial time self-stabilizing algorithms for finding dominating bipartitions, maximal independent sets, and minimal dominating sets.

The following additional definitions will be useful. A set  $S \subseteq V$  is called a *vertex cover* if every  $e \in E$  contains some member of  $S$ . We say that a vertex  $u \in V$  is *perfect* with respect to  $S$  if  $|N[u] \cap S| = 1$ . A *perfect neighborhood set* is a set  $S$  for which every vertex  $v \in V$  is either perfect, with respect to  $S$ , or is adjacent to such a vertex. Given a set  $S$  and a node  $s \in S$ , we say  $v$  is a *private neighbor* of  $s$  (with respect to  $S$ ) if  $v \in N[s] - N[S - \{s\}]$ . (The term *private neighbor* is somewhat misleading since  $v$  is allowed to equal  $s$ .) A set  $S$  is said to be *irredundant* if every member of  $S$  has a private neighbor. The following lemmas are straightforward to prove.

**Lemma 1** *Every maximal independent set is minimal dominating.*

**Lemma 2**  *$S$  is maximal independent if and only if  $S$  is independent dominating.*

**Lemma 3**  *$S$  is independent if and only if  $V - S$  is a vertex cover.*

**Lemma 4** *Every maximal independent set  $S$  is maximal irredundant.*

**Lemma 5** *Every maximal independent set is a perfect neighborhood set.*

## 2 Dominating Bipartitions

Our first self-stabilizing algorithm is shown below. Each node  $i$  has a single binary variable  $x(i)$ . The rules allow a node to change its value if all nodes in its closed neighborhood have the same value. Upon stabilization, the two sets of nodes  $\{i \mid x(i) = 0\}$  and  $\{i \mid x(i) = 1\}$  are each dominating sets, if  $G$  has no isolated nodes, thus forming a dominating bipartition. Figure 1 shows two different executions, (a)–(d) and (e)–(f), that begin with the same initial configuration.

---

**Algorithm 2.1:** DOMINATING BIPARTITION()

---

**R1:** if  $x(i) = 0 \wedge (\forall j \in N(i))(x(j) = 0)$   
then  $x(i) = 1$

**R2:** if  $x(i) = 1 \wedge (\forall j \in N(i))(x(j) = 1)$   
then  $x(i) = 0$

---

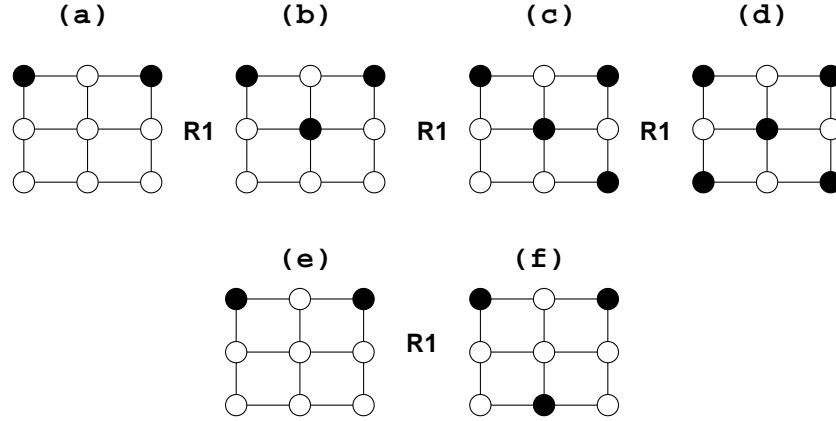


Figure 1: Starting with the initial configuration shown in (a), diagrams (a), (b), (c), (d) depict one execution of Algorithm 2.1. Nodes  $i$  for which  $x(i) = 1$  are black. Starting with the same initial configuration, diagrams (e) and (f) show an alternate execution, stabilizing in only one move. Diagrams (d) and (f) depict stable states.

**Lemma 6** *If node  $i$  ever makes a move, either **R1** or **R2**, it will never make another move, nor will any of its neighbors.*

**Proof:** Let  $i$  and  $j$  be neighbors. A node can move only if it and its neighbors all have the same value. Once  $i$  moves,  $x(i)$  and  $x(j)$  will be different, and so neither node can move.  $\square$

**Lemma 7** *Algorithm 2.1 can make at most  $n - 1$  moves.*

**Proof:** Since the network has no isolated vertices, the first node that moves must have at least one neighbor. By Lemma 6, neither of these nodes will be able to move thereafter. Also by Lemma 6, any remaining nodes can move at most once.  $\square$

**Lemma 8** *When Algorithm 2.1 stabilizes, every node labeled 0 has at least one neighbor labeled 1, and conversely, every node labeled 1 has at least one neighbor labeled 0.*

**Proof:** This is obvious.  $\square$

**Theorem 1** *In any network having no isolated nodes, Algorithm 2.1 stabilizes with a dominating bipartition in at most  $n - 1$  moves.*

**Proof:** This is immediate from Lemmas 7 and 8.  $\square$

The bound given in Theorem 1 is tight. Consider any star  $K_{1,n-1}$  in which every node is initially 1. If every leaf moves, there will there will be exactly  $n - 1$  moves.

### 3 Maximal Independent Sets

Our second self-stabilizing algorithm is a slight modification of the first, where the universal quantifier in the second rule has been replaced by an existential quantifier. Algorithm 3.1 labels the nodes in such a way that the set of nodes labeled 1 is a maximal independent set, while the set of nodes labeled 0 is a (not necessarily independent) dominating set. Thus, Algorithm 3.1 produces a dominating bipartition in which the first set is an independent dominating set. Note that we are not trying to obtain a a maximal independent set of largest cardinality.

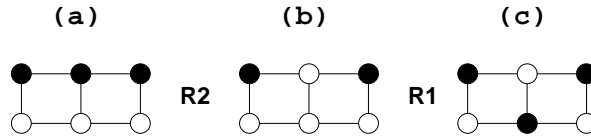


Figure 2: Diagrams (a), (b), (c) depict one execution of Algorithm 3.1. Diagrams (c) depicts a stable state.

In Algorithm 3.1 each node  $i$  has a boolean variable  $s(i)$  indicating membership in the set that we are trying to construct.

---

**Algorithm 3.1:** MAXIMALINDEPENDENT()

---

**R1:** if  $s(i) = 0 \wedge (\forall j \in N(i)) (s(j) = 0)$   
then  $s(i) = 1$

**R2:** if  $s(i) = 1 \wedge (\exists j \in N(i)) (s(j) = 1)$   
then  $s(i) = 0$

---

Figure 2 illustrates Algorithm 3.1. For purposes of this paper, we say that a node  $i$  is *independent* if

$$s(i) = 1 \wedge (\forall j \in N(i)) (s(j) = 0)$$

and that  $i$  is *dominated* if

$$s(i) = 0 \wedge (\exists j \in N(i)) (s(j) = 1).$$

By executing **R1**, a node becomes independent. By executing **R2**, a node becomes dominated. Figure 2 illustrates Algorithm 3.1.

**Lemma 9** *If every node is either independent or dominated, then the system is stable.*

**Proof:** This is obvious. □

**Lemma 10** *If the system is stable, then every node is independent or dominated.*

**Proof:** Suppose there exists a node  $i$  which is neither independent nor dominated. If  $s(i)$  is 1 and  $i$  is not independent, then  $i$  may use rule **R2**. If  $s(i) = 0$  and  $i$  is not dominated, then  $i$  may use rule **R1**. □

**Lemma 11** *The system is stable if and only if  $S = \{i \mid s(i) = 1\}$  is a maximal independent set.*

**Proof:** By Lemma 9 and Lemma 10, being stable is equivalent to every node being either independent, or dominated, which is clearly equivalent to every member of  $S$  being independent and every member of  $V - S$  being dominated. This is equivalent to  $S$  being both independent and dominating, which by Lemma 2, is equivalent to  $S$  being maximal independent. □

**Lemma 12** *Any node that becomes independent will never move again.*

**Proof:** Assume  $i$  becomes independent. Were  $i$  ever to move again, its next move would necessarily be with **R2**. But since  $i$  is independent, no neighbor of  $i$  can ever execute **R1**. Hence  $i$  will never execute **R2**. □

**Lemma 13** *A node cannot use a rule in Algorithm 3.1 twice in a row.*

**Proof:** This follows from the fact that **R1** changes  $s(i)$  from 0 to 1, and **R2** changes  $s(i)$  from 1 to 0. □

**Lemma 14** *Given any system having  $n$  nodes, and any initial state, rules **R1** and **R2** can be used at most  $2n$  times.*

**Proof:** By contradiction, suppose there is a sequence of  $2n + 1$  moves. Then there must be some node  $i$  that moves three times. From Lemma 13, it follows that during the computation,  $i$  must execute either **R1, R2, R1**, or **R2, R1, R2**. But executing **R1** causes a node to become independent, so by Lemma 12, it can never

move again, a contradiction. □

By Lemma 11 and Lemma 14 we have:

**Theorem 2** *Algorithm 3.1 finds a maximal independent set in at most  $2n$  moves.*

From Lemmas 1–5 we have:

**Corollary 1** *In any network without isolated nodes, Algorithm 3.1 identifies (i) a maximal independent set, (ii) a minimal dominating set, (iii) a perfect neighborhood set, (iv) a minimal vertex cover, and (v) a maximal irredundant set.*

We exhibit a family of graphs, each having executions of  $2n - 3$  moves, to show that the upper bound in Theorem 2 is close to tight. Consider the  $n$ -vertex graph  $K_{1,n-1}$  with a center vertex 0, and neighbors  $1, \dots, n-1$ . Initially all  $s(i)$  are 1. Each  $i$ ,  $2 \leq i \leq n-1$ , executes rule **R2** until only nodes 0 and 1 have the value one. Then node 0 executes rule **R2**, so that only node 1 has one. Finally, nodes  $2, \dots, n-1$  execute rule **R1**.

## 4 Minimal Dominating Sets

In this section we present a third self-stabilizing algorithm, Algorithm 4.1. In a graph without isolated nodes, it produces a dominating bipartition where the nodes labeled 1 define a minimal dominating set, and the nodes labeled 0 define a dominating set.

Algorithm 4.1 uses two variables. The first variable is a binary variable  $x(i)$  defining a minimal dominating set  $S = \{i \mid x(i) = 1\}$ . We will use  $S_t$  to denote this set at time  $t$ . The second variable is a pointer. By pointing to a neighbor  $j$ , written  $i \rightarrow j$ , a node  $i$  communicates to  $j$  that  $i$  is a private neighbor. That is, node  $j$  is the only node in  $S$  which currently dominates node  $i$ . The value null is used for nodes in  $S$  and nodes in  $V - S$  that are not private neighbors. We write  $i \not\rightarrow j$  to denote that  $i$  is not pointing to  $j$ , and we write  $i \not\rightarrow \text{null}$  to denote that the pointer of  $i$  is not null. Our algorithm is based on the following well-known and straightforward characterization of minimal dominating sets, whose proof can be found in [9].

**Lemma 15** *A set  $S$  is a minimal dominating set if and only if it is dominating and every  $u \in S$  has a private neighbor.*

---

**Algorithm 4.1:** MINIMAL DOMINATING SET()

---

**M1:** if  $(x(i) = 0) \wedge (\forall j \in N(i))(x(j) = 0)$   
then  $x(i) = 1$

**M2:** if  $(x(i) = 1) \wedge (\nexists j \in N(i))(j \rightarrow i) \wedge (\exists k \in N(i))(x(k) = 1)$   
then  $x(i) = 0$

**P1:** if  $(x(i) = 1) \wedge (i \nrightarrow \text{null})$   
then  $i \rightarrow \text{null}$

**P2:** if  $(x(i) = 0) \wedge (\exists \text{ unique } j \in N(i))((x(j) = 1) \wedge (i \nrightarrow j))$   
then  $i \rightarrow j$

**P3:** if  $(x(i) = 0) \wedge (\exists \text{ more than one } j \in N(i))((x(j) = 1)) \wedge (i \nrightarrow \text{null})$   
then  $i \rightarrow \text{null}$

---

Figure 3 depicts one execution of Algorithm 4.1. Rules **M1** and **M2**, which we call *membership* moves, allow nodes to change membership in the set  $S$  under construction. In particular, if a node is not a member, nor are any neighbors, then it may enter the set by **M1**. A node  $i$  that is already a member, but has a neighbor  $k$  who is also a member, may use rule **M2** to leave the set *provided* no neighbor  $j$  depends on  $i$ . Node  $i$  knows that a neighbor  $j$  is a private neighbor if  $j \rightarrow i$ .

We call rules **P1**, **P2** and **P3** *pointer* moves. They do not modify membership in the dominating set, but rather are used only to adjust pointer values so that

1. Every node  $i \in S$  has a null pointer;
2. Every node  $i \notin S$  having exactly one neighbor  $j \in S$ , points to  $j$ ;
3. Every node  $i \notin S$  having more than one neighbor  $j \in S$ , has a null pointer.

**Lemma 16** *If at time  $t$ ,  $S_t$  is not a minimal dominating set, then the system is not stable.*

**Proof:** By contradiction, suppose  $S_t$  is not minimal dominating, but the system is stable. If  $S_t$  is not a dominating set, then at least one node can make move **M1**. Hence we may assume that  $S_t$  is a dominating

set but is not minimal. Then by Lemma 15, there exists some  $i \in S_t$  that does not have a private neighbor. It follows that  $i$  must have a neighbor  $k \in S_t$ , for otherwise,  $i$  would be its own private neighbor. There must also be some  $j \in N(i)$  with  $j \rightarrow i$ , for otherwise  $i$  could make move **M2**. It must be the case that  $j \notin S_t$ , for otherwise  $j$  could execute **P1**. We know that  $j$  has only one neighbor in  $S_t$ , namely  $i$ , for otherwise  $j$  could execute **P3**. But if  $j \notin S_t$ , and  $i$  is its only neighbor in  $S_t$ , it follows that it is a private neighbor of  $i$ . This is a contradiction.  $\square$

**Lemma 17** *If a node uses **M1**, it will never make another membership move.*

**Proof:** If node  $i$  makes **M1** at time  $t$ , then none of its neighbors are in  $S_t$ . For  $i$  to later use **M2**, there must be a neighbor  $k$  for which  $x(k) = 1$ . But no  $k$  will be able to use **M1** because  $x(i) = 1$ .  $\square$

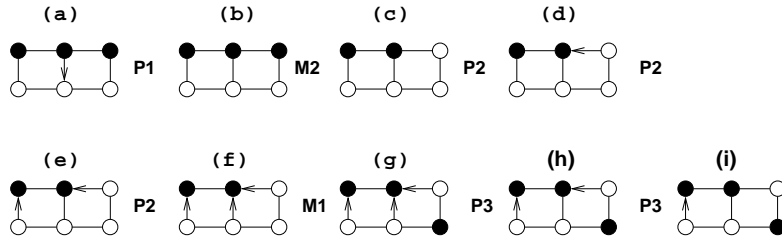


Figure 3: An execution of Algorithm 4.1. Figure (i) depicts a stable state.

**Lemma 18** *A node can make at most two membership moves.*

**Proof:** If a node's first membership move is **M1**, by Lemma 17, it will not make a membership move again. If its first membership move is **M2**, then any next membership move must be **M1**, after which, it cannot make another membership move.  $\square$

**Lemma 19** *There can be at most  $n$  consecutive pointer moves.*

**Proof:** Any pointer move by node  $i$  leaves  $i$  unprivileged. No pointer moves made by other nodes can make  $i$  privileged. Therefore, in a sequence of consecutive pointer moves, each node can move at most once.  $\square$

**Lemma 20** *The system can make at most  $(2n + 1)n$  moves.*

By Lemma 18, there are at most  $2n$  membership moves. Before and after each membership move there can be, by Lemma 19, at most  $n$  consecutive pointer moves.  $\square$

**Theorem 3** *Algorithm 4.1 produces a minimal dominating set and stabilizes in  $O(n^2)$  moves.*

**Proof:** This follows from Lemma 16 and Lemma 20.  $\square$

Algorithm 4.1, in fact, produces a dominating bipartition since the complement of a minimal dominating set always is dominating in graphs having no isolated nodes [9]. The following theorem is stated without proof.

**Theorem 4** *Algorithm 4.1 is stable if*

1.  $S_t$  is a minimal dominating set, and
2. Every private neighbor outside  $S_t$  points to its unique neighbor in  $S_t$ , and
3. All other nodes have null pointers.

The significance of Theorem 4 is that if the system is initialized to any minimal dominating set with the correct pointer settings, including minimal dominating sets that are not independent, then it will remain stable. While Algorithm 3.1 can only stabilize with an independent set, Algorithm 4.1 is capable of being stable with *any* minimal dominating set. The importance is that for some graphs no dominating set of smallest cardinality is independent. For example, consider the graph  $G$  formed by taking two stars  $K_{1,n}$ , and joining their centers by an edge. For this graph, Algorithm 3.1 will identify a set having at least  $n + 1$  nodes, but Algorithm 4.1 can be stable with the minimum cardinality having two nodes.

## 5 Concluding remarks

We have given three self-stabilizing algorithms, each of which constructs a different kind of dominating set. Algorithm 2.1 stabilizes in at most  $n - 1$  moves, but its dominating sets are not necessarily minimal. The bound  $n - 1$  is tight. Algorithm 3.1 stabilizes in at most  $2n$  moves, its dominating set is minimal, but always independent, and therefore for some graphs is never of smallest cardinality. The bound  $2n$  is almost

sharp. Algorithm 4.1 stabilizes in  $O(n^2)$  moves, is more complex, but can potentially produce *any* minimal dominating set.

Our research aims at discovering self-stabilizing algorithms for other domination-related problems. Currently we are studying maximal 2-packings, and minimal total dominating sets. These two problems seem quite difficult and may in fact be impossible within the constraints of this algorithmic model.

Algorithm 3.1 produces a bipartition of a network into an independent dominating set and a dominating set. If a network  $G$  without isolated nodes is bipartite, then, trivially, it is possible to define a bipartition of  $G$  into two independent dominating sets, simply by two-coloring  $G$ . It is interesting to note that so far we have been unable to design a self-stabilizing algorithm that can produce a two-coloring of a bipartite graph in a polynomial number of moves! We note that the self-stabilizing algorithm in [14] for two-coloring a bipartite graph has an undetermined complexity.

## References

- [1] 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** 1–7 (1995).
- [2] G. ANTONOIU AND P.K. SRIMANI, A self-stabilizing distributed algorithm to find the center of a tree graph. *Parallel Algorithms and Applications* **10** 237–248 (1997).
- [3] G. ANTONOIU AND P.K. SRIMANI, A self-stabilizing distributed algorithm to find the median of a tree graph. *J. Comput. Sys. Sci.* **58** 215–221 (1999).
- [4] S. CHANDRASEKAR AND P.K. SRIMANI, A self-stabilizing distributed algorithm for all-pairs shortest path problem. *Parallel Algorithms and Applications* **4** 125–137 (1994).
- [5] P. CHAUDHURI, A note on self-stabilizing articulation point detection. *J. Systems Arch.* **45** 1249–1252 (1999).
- [6] E.W. DIJKSTRA, Self-stabilizing systems in spite of distributed control. *Comm. ACM* **17** (11) 643–644 (1974).
- [7] E.W. DIJKSTRA, A belated proof of self-stabilization. *J. of Distributed Computing* **1** 5–6 (1986).
- [8] S. GHOSH AND M.H. KARAATA, A self-stabilizing algorithm for coloring planar graphs. *Distributed Comput.* **7** 55–59 (1993).
- [9] T.W. HAYNES, S.T. HEDETNIEMI AND P.J. SLATER, *Fundamentals of Domination in Graphs*, Marcel Dekker, New York (1998).
- [10] *Domination in Graphs: Advanced Topics*, (edited by T.W. Haynes, S.T. Hedetniemi and P.J. Slater), Marcel Dekker, New York, (1998).
- [11] S.T. HEDETNIEMI, D.P. JACOBS, AND P.K. SRIMANI, Maximal matching stabilizes in time  $O(m)$ . *Inform. Process. Lett.*, to appear.
- [12] SU-CHU HSU AND SHING-TSAAN HUANG, A self-stabilizing algorithm for maximal matching. *Inform. Process. Lett.* **43** 77–81 (1992).
- [13] S. SUR AND P.K. SRIMANI, A self-stabilizing distributed algorithm to construct BFS spanning trees of a symmetric graph. *Parallel Process. Lett.* **2** 171–179 (1992).
- [14] S. SUR AND P.K. SRIMANI, A self-stabilizing algorithm for coloring bipartite graphs. *Inform. Sci.* **69** 219–227 (1993).
- [15] GERARD TEL, Maximal matching stabilizes in quadratic time. *Information Processing Letters* **49** 271–272 (1994).
- [16] M.S. TSAI AND S.T. HUANG, A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Process. Lett.* **4** 65–72 (1994).