Self-stabilizing Algorithms for $\{k\}$-domination

Martin Gairing[1,⋆], Stephen T. Hedetniemi[2,⋆⋆], Petter Kristiansen[3], and Alice A. McRae[4]

[1] Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, 33102 Paderborn, Germany
gairing@uni-paderborn.de
[2] Department of Computer Science, Clemson University, Clemson, SC 29634, USA
hedet@cs.clemson.edu
[3] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
petterk@ii.uib.no
[4] Department of Computer Science, Appalachian State University, Boone, NC 28608, USA
aam@cs.appstate.edu

Abstract. In the self-stabilizing algorithmic paradigm for distributed computing 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. A function $f : V(G) \to \{0, 1, 2, \ldots, k\}$ is a $\{k\}$-dominating function if $\Sigma_{j \in N[i]} f(j) \geq k$ for all $i \in V(G)$. In this paper we present self-stabilizing algorithms for finding a minimal $\{k\}$-dominating function in an arbitrary graph. Our first algorithm covers the general case, where k is arbitrary. This algorithm requires an exponential number of moves, however we believe that its scheme is interesting on its own, because it can insure that when a node moves, its neighbors hold *correct* values in their variables. For the case that $k = 2$ we propose a linear time self-stabilizing algorithm.

1 Introduction

A distributed system can be modeled with a connected, undirected graph G with node set $V(G)$ and edge set $E(G)$. If i is a node in $V(G)$, then $N(i)$, its *open neighborhood*, denotes the set of nodes to which i is adjacent. Every node $j \in N(i)$ is called a *neighbor* of node i. The *closed neighborhood* of node i, is the set $N[i] = N(i) \cup \{i\}$. For a set $S \subseteq V(G)$ we define $N[S] = \bigcup_{i \in S} N[i]$. A *dominating set* is a set $S \subseteq V(G)$ for which $N[S] = V(G)$. Denote $n = |V(G)|$ the number of nodes in the graph and $m = |E(G)|$ the number of edges.

Self-stabilization is a paradigm for distributed systems that allows a system to achieve a desired, or legitimate, global state, even in the presence of faults.

⋆ Supported by the IST Program of the EU under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS).
⋆⋆ Research supported in part by NSF Grant CCR-0218495.

The concept was introduced in 1974 by Dijkstra [2], but serious work on self-stabilizing algorithms did not start until the late 1980s. (See [14, ch. 15] for an introduction to self-stabilizing algorithms.) Dolev [3] published the first book, that focuses completely on self-stabilization.

In our algorithmic model, each node executes the same self-stabilizing algorithm, maintains its own set of local variables, and changes the values of its local variables based on the current values of its variables and those of its neighbors. The contents of a node's local variables determine its *local state*. The system's *global state* is the union of all local states.

When a node changes its local state, it is said to make a *move*. Our algorithms are given as a set of rules of the form $p(i) \Rightarrow M$, where $p(i)$ is a boolean predicate, and M is a move which describes the changes to be made to one or more of the node's local variables. A node i becomes *privileged* if $p(i)$ is true. When a node becomes privileged, it may execute the corresponding move. We assume there exists a daemon, an adversarial oracle, as introduced in [2], which selects one of the privileged nodes. The selected node then makes a move. Following the serial model, we assume that no two nodes move at the same time. For our algorithm it's sufficient that adjacent nodes never move at the same time. This can be achieved using a protocol for local mutual exclusion [1].

The goal of the daemon is to keep the algorithm running as long as possible. When no further moves are possible, we say that the system is *stable* or is in a stable state. We say that a self-stabilizing algorithm is *correct* if, when the system executes the algorithm,

1) every stable state it can reach is legitimate, that is, every stable state has the desired global property, and
2) it always reaches a stable state after a finite number of moves.

Notice that it is quite possible for a correct self-stabilizing algorithm to reach a legitimate state which is not stable.

Problems that can be solved by a straightforward greedy method in the conventional algorithmic model often require a far more clever approach in the self-stabilizing model. For example, finding a maximal matching (i.e. a set of disjoint edges that cover all remaining edges) in a graph is trivial: starting with an empty set of edges, keep adding another disjoint edge to the set as long as one exists. To describe and prove the correctness of a self-stabilizing algorithm for this same problem takes considerably more effort [11, 13, 10].

2 $\{k\}$-domination

For a graph G, let $f : V(G) \to \{0, 1, \ldots, k\}$ be a function from the node set $V(G)$ into the set of integers $\{0, 1, \ldots, k\}$, and let $f(S) = \sum_{v \in S} f(v)$ for any subset $S \subseteq V(G)$. Such a function is a $\{k\}$-*dominating function* [8, 4, 5], if for every $i \in V(G)$ we have $f(N[i]) \geq k$. A $\{k\}$-dominating function f is *minimal* if there does not exist a $\{k\}$-dominating function f' with $f' \neq f$ and $f'(i) \leq$

$f(i), \forall i \in V(G)$. Or equivalently, for every node i, if $f(i) \neq 0$, then there is a neighbor $j \in N[i]$ with $f(N[j]) = k$.

The general idea of $\{k\}$-domination is that every node in the network needs a minimum amount of a given resource, for example k units of it. It then becomes a resource allocation problem to locate a minimum quantity of this needed resource among the nodes so that each node can gain access to at least k units of this resource within its closed neighborhood. The node must either supply itself with this resource, if it does not exist among its neighbors, or it can use the resources of its neighbors. Examples of this resource might include a minimum number of firefighters, ambulances, or emergency vehicles, or a minimum number of CPU cycles, memory or servers. Of course, in the most general setting, all nodes have different resource needs, say as a function of their size, population, etc. Thus, we can associate with each node i, a minimum resource requirement r_i. We then seek an allocation of resources, at minimum cost, such that the total amount of resources assigned to each closed neighborhood $N[i]$ is at least r_i. In the definition of $\{k\}$-domination it is assumed that all nodes have the same resource requirement, i.e. k.

The concept of a $\{k\}$-dominating function gives rise to the following decision problem:

$\{k\}$-DOMINATING FUNCTION
INSTANCE: A graph G and a positive integer l.
QUESTION: Does G have a $\{k\}$-dominating function f, with $f(V(G)) \leq l$?

In the reduction below we use the following problem, first shown to be $\mathcal{NP}$-complete by Karp in [12]:

MINIMUM COVER
INSTANCE: A set $X = \{x_1, x_2, \ldots, x_n\}$ of n elements, a collection $S = \{S_1, S_2, \ldots, S_m\}$ of m subsets of X, and a positive integer $q \leq |S|$.
QUESTION: Does S contain a cover for X of size q or less, i.e., a subset $S' \subseteq S$ with $|S'| \leq q$, such that every element of X belongs to at least one member of S'?

Theorem 1. $\{k\}$-DOMINATING FUNCTION *is $\mathcal{NP}$-complete for all $k \geq 1$.*

Proof. $\{k\}$-DOMINATING FUNCTION is clearly in $\mathcal{NP}$. We reduce from MINIMUM COVER.

Given an instance (X, S, q) of MINIMUM COVER, we create an instance (G, l) of $\{k\}$-DOMINATING FUNCTION in the following manner: for each $x_i \in X$, there is an element component G_i consisting of k paths on three nodes. We label the nodes of path j, for $1 \leq j \leq k$: $a_{i,j}$, $b_{i,j}$, $c_{i,j}$. In addition, one node d_i is connected to each of the c-nodes $c_{i,1}, \ldots, c_{i,k}$. For each subset $S_{i'} \in S$ there is a subset node $u_{i'}$ which is is connected to all of the $a_{i,j}$ nodes in the element components corresponding to the elements in $S_{i'}$. All subset nodes are in turn connected to a common K_2, with nodes labeled v and w, by adding edges from all subset nodes $u_{i'}$ to v. The transformation is completed by taking $l = |X| \cdot k^2 + q + k$.

Assume T is a set cover with $|T| \leq q$. We construct a $\{k\}$-dominating function as follows: for the element components G_i we let $f(a_{i,j}) = 0$, $f(b_{i,j}) = k-1$, $f(c_{i,j}) = 1$, and $f(d_{i,j}) = 0$. For the subset nodes we let $f(u_{i'}) = 1$ if $S_{i'} \in T$, and $f(u_{i'}) = 0$ if $S_{i'} \notin T$. Finally we let $f(v) = k$ and $f(w) = 0$.

Under f, all b-, c-, and d-nodes will have neighborhood sums of k. The a-nodes will each be adjacent to a b-node with value $k-1$, and at least one u-node with value 1. The remaining nodes all have node v in their neighborhood, and will therefore have neighborhood sums of k. Note that, for each element component G_i we have $f(G_i) = k^2$. The f-values for the subset nodes will sum to at most q, and $f(v) = k$. This implies that $f(V(G)) \leq |X| \cdot k^2 + q + k$, and, therefore f is a $\{k\}$-dominating function of weight less than or equal to l.

Given a $\{k\}$-dominating function f for G, with $f(V(G)) \leq l$, we modify f as follows: 1) If $f(v) \neq k$, change $f(v)$ and $f(w)$ so that $f(v) = k$ and $f(w) = 0$. It is easy to see that the modified function will still be a $\{k\}$-dominating function with $f(V(G)) \leq l$. 2) Examine each element component G_i, one at a time. Note that the f-values of every abc-path within the component must sum up to at least k, otherwise the b-node of the path will not be $\{k\}$-dominated. Therefore, $f(G_i) \geq k^2$, for every element component G_i. There are two cases to consider: i) $f(G_i) = k^2$, and ii) $f(G_i) > k^2$. In case i) node d_i, which is not part of any abc-path must have $f(d_i) = 0$. Therefore the sum of the f-values for the b-, and c-nodes of every abc-path must be k, otherwise the c-nodes will not be $\{k\}$-dominated. It therefore follows that $f(a_{i,j}) = 0$, for all j. In case ii) we reassign the f-values of all the nodes in G_i, so that $f(a_{i,j}) = 0$, $f(b_{i,j}) = k-1$, and $f(c_{i,j}) = 1$, for every j, and $f(d_i) = 0$. We then choose any u-node connected to G_i, say $u_{i'}$, and set $f(u_{i'}) = 1$ if $f(u_{i'}) = 0$. The modified f is still a $\{k\}$-dominating function, and the weight of f has not increased. With the modified f-function we find a set cover T by taking $T = \{S'_i : f(u_{i'}) \geq 1\}$.

Since $f(V(G)) \leq l = |X| \cdot k^2 + k + q$, we know that the f-values of the u-nodes sum up to at most q (remember $f(v) = k$ and $f(G_i) = k^2$), therefore there are at most q u-nodes with f-value greater than zero. Each element component G_i will have one c-node, say $c_{i,j}$, with $f(c_{i,j}) \geq 0$, otherwise d_i will not be $\{k\}$-dominated. Therefore $f(b_{i,j}) < k$. Consider node $a_{i,j}$. We know $f(a_{i,j}) + f(b_{i,j}) < 0 + k$, so $a_{i,j}$ must be adjacent to some u-node $u_{i'}$ with $f(u_{i'}) > 0$. Every element component must therefore be adjacent to a u-node, say $u_{i'}$, with $f(u_{i'}) > 0$, the subsets corresponding to those u-nodes will constitute a set cover. □

3 Minimal $\{k\}$-dominating function

In this section we present a self-stabilizing algorithm for finding a minimal $\{k\}$-dominating function in an arbitrary graph. The algorithm is based on a self-stabilizing algorithm for maximal 2-packing [6]. We assume that every node has a distinct identifier, or *ID*, and that there exists a total ordering on the IDs. Each node i has a local variable f, storing the function value associated with node i, a local variable σ storing a local copy of the value $f(N[i])$ and a pointer. We use the notation $i \to j$ and $i \to NULL$ to denote that the pointer of i points

to j and that the pointer of i points to $NULL$, respectively. A node i can also point to itself, indicated by $i \to i$. For a node i we say that $\sigma(i)$ is *correct*, if $\sigma(i) = f(N[i])$. The algorithm is identically stored and executed in each node i. For simplicity we define the following shorthands:

- $predf(i) = f(N[i]) < k \vee (f(i) > 0 \wedge f(N[i]) > k \wedge \forall j \in N(i) : \sigma(j) > k)$
- $minn(i) = min\{j | j \in N(i) \wedge j \to j\}$ and $min(\emptyset) = NULL$.

The predicate $predf(i)$ is used to indicate whether node i should change its f-value or not. For a node i, $predf(i)$ is true if either $f(N[i]) < k$ or from the view of i, $f(i)$ can still be decreased, preserving the property that $f(N[j]) \geq k$ for all $j \in N[i]$. We define $minn(i)$ to be the minimum ID of a node in the open neighborhood of i, that points to itself. If no such node exists, $minn(i) = NULL$.

Algorithm 3.1: MINIMAL $\{k\}$-DOMINATING FUNCTION

local $f(\_) \in \{0, 1, \ldots, k\}$;
$\sigma(\_)$ integer;
$\_ \to \_$ pointer;

ADD: if $predf(i) \wedge i \to NULL \wedge \forall j \in N(i) : j \to NULL$
then $\begin{cases} i \to i; \\ \sigma(i) = f(N[i]); \end{cases}$

UPDATE: if $\sigma(i) \neq f(N[i]) \vee (i \not\to i \wedge i \not\to minn(i))$
then $\begin{cases} i \to minn(i); \\ \sigma(i) = f(N[i]); \end{cases}$

RETRACT: if $i \to i \wedge \exists j \in N(i) : (j \to l \wedge l < i)$
then $\begin{cases} i \to minn(i); \\ \sigma(i) = f(N[i]); \end{cases}$

INCREASE: if $i \to i \wedge \forall j \in N(i) : j \to i \wedge f(N[i]) \leq k$
then $\begin{cases} f(i) = k - f(N(i)); \\ \sigma(i) = f(N[i]); \\ i \to NULL; \end{cases}$

DECREASE: if $i \to i \wedge \forall j \in N(i) : j \to i \wedge f(N[i]) > k$
then $\begin{cases} \sigma(i) = f(N[i]); \\ f(i) = f(i) - min_{j \in N[i]}\{(\sigma(j) - k), 0\}; \\ \sigma(i) = f(N[i]); \\ i \to NULL; \end{cases}$

The rules ADD, UPDATE and RETRACT are used to achieve mutual exclusion for the INCREASE and DECREASE moves and to make sure that whenever a node INCREASEs or DECREASEs, this move is based upon correct

σ-variables in its neighborhood. We will use the term *p-move* for an ADD, UPDATE or RETRACT and *f-move* for an INCREASE or DECREASE.

A node i that has to change its f-value ($predf(i)$ is true) tries to ADD. It can make this ADD move if all of its neighbors (including i) point to $NULL$. When making an ADD move, node i sets its pointer to itself, indicating that i wants to change its f-value. The next time that node i gets scheduled it either changes its f-value (by an f-move) or it RETRACTs. The f-move is made if all neighbors of i point to i. When executing this f-move, node i updates $f(i)$ and sets its pointer back to $NULL$. The RETRACT move is made, if i has a neighbor that points to a node with lower ID than the ID of i. A RETRACT move sets the pointer of i to $minn(i)$, that is, the neighbor with smallest ID that points to itself (if it exists) or to $NULL$. A node i where $\sigma(i)$ is not correct, or a node that does not point to itself or to $minn(i)$ can UPDATE by setting its pointer to $minn(i)$. In every move node i also recalculates $\sigma(i)$.

We prove convergence of Algorithm 3.1 in two steps. First, we give an upper bound on the total number of p-moves that the daemon can make without making an f-move. We then show that the number of f-moves for each node is bounded by a constant. For the first step we assume that $predf(i) = TRUE$ for every node i. It is easy to see that this gives the daemon the most power, since otherwise all nodes j with $predf(j) = FALSE$ are not privileged by ADD.

Lemma 1. *The following holds in between two consecutive INCREASE or DECREASE moves:*
(a) Between two successive UPDATEs by node i, there must be a RETRACT or ADD by a neighbor of i.
(b) Between two successive ADDs by node i, there must be a RETRACT by a lower numbered node.

Proof. (a) After the first UPDATE, i points to $minn(i)$ and $\sigma(i) = f(N[i])$. For another UPDATE to occur, i must not point to $minn(i)$. This means that $minn(i)$ has changed.

(b) When node i ADDs, all of its neighbors point to $NULL$. For i to RETRACT at least one of its neighbors, say j, must point to a lower-numbered node, say z. By the time i ADDs again, j must have changed its pointer back to $NULL$. If $j = z$, then that means that z has RETRACTed. On the other hand, if $j \neq z$, then since $j \to NULL$ previously, at the moment that j changes its pointer to z, the node z points to itself. At the moment that j changes its pointer back to $NULL$, the node z must not point to itself. That is z has RETRACTed. □

Assume that the nodes are numbered from 1 to n. Let $a(i)$ denote the number of times that node v_i executes an ADD rule and let $r(i)$ be the number of RETRACTs of node v_i. Obviously, ADDing and RETRACTing are closely linked. Indeed, $a(i) - 1 \leq r(i) \leq a(i) + 1$.

Lemma 2. *In between two INCREASEs or DECREASEs in the graph the number of ADDs for a node j is bounded by $a(j) \leq 2^j - 1$.*

Proof. By induction on j. Base case: Node v_1 can ADD only once by the above lemma.

For the general case, Lemma 1 says that between consecutive ADDs for node v_j, there must be a RETRACT of one of $v_1, \ldots, v_{j-1}$. This means that:

$$a(j) \leq 1 + \sum_{i<j} r(i) \leq 1 + \sum_{i<j} (1 + a(i)) \leq 2^j - 1,$$

where the last inequality follows from the inductive hypothesis. □

Lemma 3. *At most $(n+1) \cdot 2^{n+2}$ ADDs, UPDATEs and RETRACTs can be made without executing DECREASE or INCREASE.*

Proof. By Lemma 2, the total numbers of ADDs and RETRACTs are both at most 2^{n+1}. By Lemma 1, the total number of UPDATEs is at most n times the total number of ADDs and RETRACTs combined. □

We now show some properties that hold prior to, or just after an f-move.

Lemma 4. *If a node i makes an f-move for the second time, this move is based upon correct values of $\sigma(j)$ for all neighbors $j \in N[i]$.*

Proof. The first time a node i made an f-move all neighbors $j \in N[i]$ were pointing at i. After i INCREASEd or DECREASEd it sets its pointer back to $NULL$. In order for i to make an f-move again it first has to ADD. This only happens when all its neighbors $j \in N(i)$ are pointing to $NULL$. Then each neighbor j has to change its pointer back to i. At every pointer change j will set $\sigma(j) = f(N[j])$. Since $j \to i$ no other neighbor $z \in N(j) \setminus \{i\}$ can INCREASE or DECREASE. Therefore $f(N[j])$ will not change as long as $j \to i$ and i does not INCREASE or DECREASE. The lemma follows. □

Lemma 5. *After the first INCREASE or DECREASE of node i, for the rest of the execution of the algorithm $f(N[i]) \geq k$.*

Proof. When node i INCREASEs it will set $f(i)$ such that $f(N[i]) = k$. It's easy to see, that after a DECREASE $f(N[i]) \geq k$. After an INCREASE or DECREASE move, $\sigma(i) = f(N[i])$. By the proof of Lemma 4 it follows that a later DECREASE of some neighbor $j \in N(i)$ will not push $f(N[i])$ below k. □

To show that Algorithm 3.1 stabilizes we now prove that every node can make at most two f-moves.

Lemma 6. *The number of combined INCREASEs and DECREASEs for each node is at most 2.*

Proof. The first f-move that a node, say i, can make is either INCREASE or DECREASE. By Lemma 5 it follows that from there on $f(N[i]) \geq k$ and therefore the predicate for an INCREASE does not hold. Lemma 4 says, that from the second f-move on, when i DECREASEs again, for all neighbors $j \in N(i)$: $\sigma(j) = f(N[j])$. This implies, that after a DECREASE of node i either $f(i) = 0$ or $\exists j \in N[i] : f(N[j]) = k$ and $f(N[z]) \geq k, \forall z \in N[i] \setminus \{j\}$. In either case i will not DECREASE again. □

Lemma 7. *When the algorithm stabilizes, the function f represents a minimal $\{k\}$-dominating function.*

Proof. In a stable configuration no node points to itself. Assume that there are nodes that point to themselves and let i be such a node with smallest ID. Either there is a node $j \in N(i)$ with $j \not\to i$, in this case j can UPDATE, or $\forall j \in N(i) : j \to i$, then i can INCREASE or DECREASE. Since in a stable state no node can ADD and by the previous statement $predf(i)$ has to be $FALSE$ for every node i. By the definition of $predf$, f is then a minimal $\{k\}$-dominating function. □

Theorem 2. *Algorithm 3.1 stabilizes with a $\{k\}$-dominating function in at most $(2n+1) \cdot (n+1) \cdot 2^{n+2}$ moves.*

Proof. This follows directly from Lemma 3, 6 and 7. □

4 Minimal {2}-dominating function

Since Algorithm 3.1 requires an exponential number of moves, we will now present a polynomial time algorithm that finds a minimal $\{2\}$-dominating function in a general graph. We don't consider a $\{1\}$-dominating function, because a minimal $\{1\}$-dominating function is equivalent to a *minimal dominating set* (see [9] for a self-stabilizing algorithm). For the case $k = 2$ we propose the following algorithm. Each node i has a variable f storing the function value associated with node i. The algorithm is identically stored and executed in each node i.

Algorithm 4.1: MINIMAL $\{2\}$-DOMINATING FUNCTION

local $f(\_) \in \{0, 1, 2\}$;

R1: if $f(N(i)) = 0 \wedge f(i) \neq 2$
then $f(i) = 2$;

R2: if $f(i) = 0 \wedge f(N(i)) = 1$
then $f(i) = 1$;

R3: if $f(i) = 1 \wedge |\{j \in N(i) : f(j) > 0\}| \geq 2$
then $f(i) = 0$;

R4: if $f(i) = 2 \wedge f(N[i]) > 2$
then
if $f(N(i)) = 1$
then $f(i) = 1$;
else $f(i) = 0$;

The next two lemmas show that Algorithm 4.1 stabilizes with a $\{2\}$-dominating function.

Lemma 8. *If the system reaches a stable state, then the state is legitimate.*

Proof. If rules **R1** and **R2** do not apply, then f must be a $\{2\}$-dominating function; and if rules **R3** and **R4** do not apply, then f must be a minimal $\{2\}$-dominating function. □

Lemma 9. *If the system is in an illegitimate state, then there exists at least one node that can make a move.*

Proof. Assume the system is in an illegitimate state. There are two cases to consider: 1) the function f is not a $\{2\}$-dominating function, and 2) the function f is a $\{2\}$-dominating function, but not a minimal $\{2\}$-dominating function.

In case 1) there will exist a node i with $f(N[i]) < 2$. This node can execute rule **R1** if $f(N(i)) = 0$, rule **R2** if $f(i) = 0$ and $f(N(i)) = 1$.

In case 2) there will exist a node i with $f(N[i]) > 2$ and $\forall j \in N(i) : f(N[j]) > 2$. Either this node can execute rule **R3**, if $f(i) = 1$ and node i has at least two neighbors with positive f-value; or there will exist a node j adjacent to node i having $f(j) = 2$, and all other nodes adjacent to node i have f-value 0, in this case node j can execute rule **R4**. In either case, therefore, the non-minimal 2-dominating function does not constitute a stable state. □

We proceed by showing that any node executing rule **R1** can make no further moves, and that the remaining nodes can make only a finite number of moves.

Lemma 10. *If a node executes rule **R1** it will never make another move.*

Proof. A node i can only execute rules **R1** if $f(N(i)) = 0$. After i has executed rule **R1** no node in $N(i)$ can ever become privileged and make a move, therefore node i will also remain unprivileged. □

Lemma 11. *A node can only execute rule **R4** once.*

Proof. A node can only execute rule **R4** if $f(i) = 2$. If a node is to execute rule **R4** twice it must execute **R4**, **R1**, **R4**. But any node executing rules **R1** can never make another move. □

Let us represent the execution of Algorithm 4.1 as a sequence of moves $M_1, M_2, M_3, \ldots$, in which M_k denotes the k-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. At time t we let

$$f_t = f(V(G)) \text{ , and}$$
$$p_t = |\{e = ij \in E(G) : f(i) > 0 \wedge f(j) > 0\}| \text{ ,}$$

where f_t is the total weight of the function f, and p_t is the number of edges, both of whose nodes have strictly positive function values.

Observation 1 *If rule **R2** is executed at time t, then $f_{t+1} = f_t + 1$ and $p_{t+1} = p_t + 1$.*

Observation 2 *If rule **R3** is executed at time t, then $f_{t+1} = f_t - 1$ and $p_{t+1} \leq p_t - 2$.*

Observation 3 *If rule **R4** is executed at time t, then $f_t - 2 \leq f_{t+1} \leq f_t - 1$ and $p_{t+1} \leq p_t$.*

Theorem 3. *Algorithm 4.1 finds a minimal $\{2\}$-dominating function in at most $3n + 2m$ moves.*

Proof. Assume that the algorithm runs forever. In an infinite execution sequence there must be two states S_n and S_m, such that $S_n = S_m$ and all moves M_l, $n \leq l \leq m$, are made by executing rules **R2** and **R3**. We cannot have $S_n = S_m$ if M_l is made by executing rule **R1**, and rule **R4** can never be executed more than n times. When $S_n = S_m$ we obviously have $f_n = f_m$ and $p_n = p_m$. Let x be the number of times rule **R2** is executed and y be the number of times rule **R3** is executed. We get the following pair of equations:

$$f_m = f_n + x - y = f_n \tag{1}$$

$$p_m = p_n + x - 2y = p_n \tag{2}$$

It follows from (1) that we must have $x = y$, and it follows from (2) that we must have $x = 2y$, a contradiction. Since this set of equations has no positive solutions we have that $S_n \neq S_m$ for all $n \neq m$, thus the algorithm terminates.

To prove the desired time bound we first observe that for all states S_t, we have $0 \leq f_t \leq 2n$, and $0 \leq p_t \leq m$.

We know from Lemma 10 that rule **R1** can be executed at most n times in total. Observations 1 and 3, and Lemma 11 imply that rules **R2** and **R4** can be executed at most $2n$ times in conjunction with each other, as rule **R4** can be executed at most n times. And it follows from Observations 1 and 2 that rules **R2** and **R3** can be executed at most $2m$ times in conjunction with each other; since for two states S_n and S_m with $f_n = f_m$, and all intermediate moves made by executing rules **R2** and **R3**, we must have $p_m \leq p_n - 1$. This implies that that the algorithm stabilizes in at most $3n + 2m$ moves. □

Corollary 1. *If G is planar, then Algorithm 4.1 finds a minimal $\{2\}$-dominating function in $O(n)$ moves.*

5 Concluding remarks

In this paper we have presented self-stabilizing algorithms for finding a minimal $\{k\}$-dominating function in general graphs. Algorithm 3.1 can be used for an arbitrary k, but requires an exponential number of moves. We believe that the scheme of Algorithm 3.1 is interesting on its own, since it can be used to ensure

that certain moves are made based on neighbor variables that are correct. For instance this scheme can be used by a node to access variables of nodes that are at distance 2. With this property more complicated functions can be implemented.

Our second algorithm is dedicated to the case where $k = 2$. Here we have given a more efficient algorithm that stabilizes in at most $3n + 2m$ moves. It is interesting to consider whether a linear self-stabilizing algorithm exists for $\{3\}$-domination, and, indeed, for any fixed $\{k\}$-domination.

Subsequent to the development of Algorithm 3.1 for finding a minimal $\{k\}$-dominating function in an arbitrary graph, the authors of [7] designed a self-stabilizing algorithm for finding a minimal total dominating set in an arbitrary graph. At the end of this paper [7], the authors suggested that a self-stabilizing algorithm for $\{k\}$-domination can be constructed from the algorithm for finding a minimal total dominating set in [7]. However, this adaptation would involve the use of k pointers, and would be somewhat more complicated than Algorithm 3.1 presented in this paper.

Acknowledgment: The authors are grateful to the referees for making several suggestions for improving this paper. In particular, one referee suggested a simplified version of Algorithm 4.1, as follows:

$$\textbf{R1: if } f(i) \neq g(i) := \max\{2 - f(N(i)), 0\} \textbf{ then } f(i) = g(i);$$

Although this algorithm is different than Algorithm 4.1, it still produces a minimal $\{2\}$-dominating function and can be shown to stabilize in a linear number of moves.

References

1. J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer LNCS:1914*, pages 223–237, 2000.
2. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.
3. S. Dolev. *Self-stabilization.* MIT Press, 2000.
4. G. S. Domke. *Variations of Colorings, Coverings, and Packings of Graphs.* PhD thesis, Clemson University, 1988.
5. G. S. Domke, S. T. Hedetniemi, R. C. Laskar, and G. Fricke. Relationships between integer and fractional parameters of graphs. In Y. Alavi, G. Chartrand, O. R. Ollermann, and A. J. Schwenk, editors, *Graph Theory, Combinatorics, and Applications, Proceedings of the Sixth Quadrennial Conference on the Theory and Applications of Graphs (Kalamazoo, MI, 1988)*, volume 2, pages 371–387. Wiley, 1991.
6. M. Gairing, R. M. Geist, S. T. Hedetniemi, and P. Kristiansen. A self-stabilizing algorithm for maximal 2-packing. Technical Report 230, Department of Informatics, University of Bergen, 2002.
7. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. A self-stabilizing distributed algorithm for minimal total domination in an arbitrary system graph.

In *Proceedings of the 8th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'03) in conjunction with IPDPS'03*, 2003.
8. T. W. Haynes, S. T. Hedetniemi, and P. J. Slater. *Fundamentals of Domination in Graphs*. Marcel Dekker, 1998.
9. S. M. Hedetniemi, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithm for minimal dominating and maximal independent sets. *Comput. Math. Appl.*, to appear.
10. S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Maximal matching stabilizes in time $O(m)$. *Information Processing Letters*, to appear.
11. Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.
12. R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thathcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.
13. G. Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.
14. G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, second edition, 2000.