International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 3, Number 1, pages 116–136, January 2013

Self-Stabilizing Small k-Dominating Sets

Ajoy K. Datta, Lawrence L. Larmore
School of Computer Science, UNLV
Las Vegas, USA

Stéphane Devismes, Karel Heurtefeux, Yvan Rivierre
Verimag Lab., Université Joseph Fourier
Grenoble, France

Received: January 11, 2012
Revised: June 5, 2012
Revised: June 22, 2012
Accepted: July 19, 2012
Communicated by Yasuaki Ito

Abstract

A self-stabilizing algorithm, after transient faults hit the system and place it in some arbitrary global state, causes the system to recover in finite time without external (*e.g.*, human) intervention. In this paper, we give a distributed asynchronous silent self-stabilizing algorithm for finding a minimal k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes in an arbitrary identified network of size n. We give a transformer that allows our algorithm to work under an unfair daemon, the weakest scheduling assumption. The complexity of our solution is $O(n)$ rounds and $O(\mathcal{D}n^3)$ steps using $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, where $\mathcal{D}$ is the diameter of the network and N is an upper bound on n.

Keywords: Distributed systems, self-stabilization, k-dominating sets, k-clustering

1 Introduction

Consider a simple undirected connected network $G = (V, E)$, where V is a set of processes and E a set of edges. For any processes p and q, we define $||p, q||$, the *distance* from p to q, to be the length of the shortest path in G from p to q. Given a non-negative integer k, a subset of processes D is a *k-dominating set* of G if every process that is not in D is at distance at most k from a process in D.

Building a *k-dominating set* in a network is useful because it allows the network to be partitioned into k-clusters. A *k-cluster* of G is defined to be a set $C \subseteq V$, together with a designated process $\mathit{Clusterhead}(C) \in C$, such that each member of C is within distance k of $\mathit{Clusterhead}(C)$. We define a *k-clustering* of a graph to be a partitioning of the graph into distinct k-clusters. The set of clusterheads of a k-clustering is a k-dominating set; conversely, if D is a k-dominating set, a k-clustering is obtained by having every process choose the closest member of D as its clusterhead, ties being resolved arbitrarily.

[0]This work has been partially supported by the ANR project *ARESA*2.

A major application of k-*clustering* is in implementation of an efficient routing scheme. For example, we could use the rule that a process that is not a clusterhead communicates only with processes in its own cluster, and that clusterheads communicate with each other *via* virtual "super-edges," implemented as paths in the network.

Ideally, we would like to find a *minimum* k-dominating set, namely a k-dominating set of the smallest possible cardinality. However, this problem is known to be $\mathcal{NP}$-hard [19]. We instead consider the problem of finding a *minimal* k-dominating set, *i.e.,* a k-dominating set which has no k-dominating proper subset.

Minimality does not guarantee that a k-dominating set is small. See, for example, Figure 1. The singleton $\{v_0\}$ is a minimal 1-dominating set. However, the set of gray processes is also a minimal 1-dominating set. In this paper, we address this problem by giving a self-stabilizing algorithm that builds a minimal k-dominating set whose size is bounded by $\lceil \frac{n}{k+1} \rceil$, where n is the size of the network.

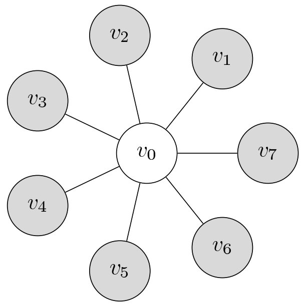

Figure 1: Example of a large, but minimal, 1-dominating set

1.1 Related Work

Self-stabilization [15, 16] enables an algorithm to withstand transient faults in a distributed system. A self-stabilizing algorithm, after transient faults place it in some arbitrary global state, causes the system to recover in finite time without external (*e.g.*, human) intervention.

There exist several asynchronous self-stabilizing distributed algorithms for finding a k-dominating set of a network, *e.g.*, [11, 10, 6]. All these algorithms work under an unfair daemon. The solution in [11] stabilizes in $O(k)$ rounds using $O(k \log n)$ space per process. The solution in [10] stabilizes in $O(n)$ rounds using $O(\log n)$ space per process. The solution given in [6] stabilizes in $O(kn)$ rounds using $O(k \log n)$ space per process. Only the algorithm of [10] builds a k-dominating set that is minimal. Moreover, none of these solutions guarantees to output a small k-dominating set. There are several self-stabilizing solutions that compute a minimal 1-dominating set, *e.g.*, [29, 22]. However, the solution for 1-dominating sets does not scale up well to k-dominating sets. In particular it does not maintain interesting bounds on the size of the computed dominating set.

There exist several *non self-stabilizing* distributed solutions for finding a k-dominating set in a network [26, 25, 1, 18, 28]. Deterministic solutions given in [1, 18] are designed for *asynchronous mobile ad hoc* networks, *i.e.*, they assume networks with a Unit Disk Graph (UDG) topology. The time and space complexities of the solution in [1] are $O(k)$ and $O(k \log n)$, respectively. The solution given in [18] is an approximation algorithm whose time complexity is $O(k)$ times that of the optimal solution. The time and space complexities of the distributed algorithm in [18] are not given. In [26], the authors consider the problem of deterministically finding a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. Their solution assumes a synchronous system and has $O(k \log^* n)$ time complexity. However, the authors missed one special case, which unfortunately invalidates their proof for some networks. The same flaw is present in some subsequent papers [25, 27]. Ravelomanana [28] gives a randomized algorithm designed for synchronous UDG networks whose time complexity is $O(\mathcal{D})$ rounds.

All previous non self-stabilizing solutions can be transformed into self-stabilizing ones using some *transformers* [23, 9]. However, the transformed self-stabilizing solutions are expected to be inefficient, both in time and space. These transformers use snapshots and similar techniques.

1.2 Contributions

In this paper, we give a deterministic, distributed, asynchronous, silent, and self-stabilizing algorithm for finding a minimal k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes in an arbitrary identified network, *i.e.,* a network where processes have unique identifiers.

We first consider the upper bound on the size of minimum k-dominating sets given in [26]. We show that the proof given in [26] missed a case, and give a correction that does not change the bound.

Next, we give an asynchronous silent self-stabilizing algorithm, called $\mathcal{SMDS}(k)$, for finding a minimal k-dominating set of small size. To simplify the design of our algorithm, we define it to be a composition of three layers. The first two layers together compute a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. The resulting k-dominating set may not be minimal. We apply the algorithm given in [10] in the last layer to remove processes from D until we obtain a minimal k-dominating set. The three layer composed algorithm is proven assuming a weakly fair daemon. The solution stabilizes in $O(n)$ rounds using $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, where N is an upper bound on n, the size of the network.

We then give a general method to efficiently transform a self-stabilizing weakly fair algorithm into a self-stabilizing algorithm working under an unfair daemon. The given transformer has several advantages over the previous solutions. (1) It preserves silence. (2) It does not degrade the round complexity or the memory requirements of the input algorithm. (3) It algorithms with step complexity ($O(\mathcal{D}n(R + n^2))$, where R is the stabilization time of the input algorithm in rounds). For example, using this method, the transformed version of $\mathcal{SMDS}(k)$ stabilizes in $O(n^3\mathcal{D})$ steps, where $\mathcal{D}$ is the diameter of the network.

Finally, we analyze, using simulations, the size of the k-dominating set computed by our algorithm. Simulation results show that the average size of the k-dominating sets we obtain from our algorithm is significantly smaller than the upper bound. In particular, we observe a noticeable decrease in the size after the minimization performed by the last layer.

1.3 Roadmap

In the next section, we give the computational model used in this paper. In Section 3, we give a counterexample for the proof of the upper bound given in [26], as well as a correction. In Section 4, we define a composition technique derived from that of Herman, [20]. This technique is used to build our self-stabilizing algorithm, Algorithm $\mathcal{SMDS}(k)$, which is presented and proven in Section 5. In Section 6, we show how to transform Algorithm $\mathcal{SMDS}(k)$ to obtain a solution that works under an unfair daemon. We present our simulation results in Section 7, and we make concluding remarks in Section 8.

2 Preliminaries

2.1 Computational Model

We consider a network to be an undirected simple connected graph $G = (V, E)$, where V is a set of n processes and E a set of bidirectional links. Processes are assumed to have distinct identifiers. In the this paper, we make no distinction between a process and its identifier, that is, the identifier of process p is simply denoted by p. If b bits are used to store each identifier, then the space complexity of our algorithm will be $\Omega(b)$ per process, but henceforth, as is commonly done in the literature, we will assume that $b = O(\log n)$.

We assume the *shared memory model* of computation, introduced by Dijkstra [15]. In this model, a process p can read its own variables and that of its neighbors, but can write only to its own variables. Let $\mathcal{N}_p$ denote the set of neighbors of p.

Each process operates according to its (local) *program*. A *distributed algorithm* $\mathcal{A}$ is a collection of n *programs*, each operating on a single process. In the following, we will denote the local program of process p in the distributed algorithm $\mathcal{A}$ by $\mathcal{A}(p)$. The program of each process is a set of actions of the following form:

$$\langle label \rangle \;::\; \langle guard \rangle \;\longrightarrow\; \langle statement \rangle.$$

Labels are only used to identify actions in the discussion. The *guard* of an action in the program of a process p is a Boolean expression involving the variables of p and its neighbors. The *statement* of an action of p updates one or more variables of p. An action can be executed only if it is *enabled*, *i.e.,* its guard evaluates to TRUE. A process is said to be *enabled* if at least one of its actions is enabled. The *state* of a process in the (distributed) algorithm $\mathcal{A}$ is defined by the values of its variables in $\mathcal{A}$. A *configuration* of $\mathcal{A}$ consists of one state for each process. We denote by $\gamma(p)$ the state of process p in configuration γ.

We use the *composite atomicity* [16] model. Let $\mapsto$ be the binary relation over configurations of $\mathcal{A}$ such that $\gamma \mapsto \gamma'$ if and only if it is possible for the network to change from configuration γ to configuration γ' in one step of $\mathcal{A}$. An *execution* of $\mathcal{A}$ is a maximal sequence of its configurations $e = \gamma_0\gamma_1 \dots \gamma_i \dots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration, a configuration in which no action of $\mathcal{A}$ is enabled at any process. Each step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* processes executing an action. The evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step.

We assume that each step from a configuration to another is driven by a *scheduler*, also called a *daemon*. If one or more processes are enabled, the scheduler selects at least one of these enabled processes to execute an action. The *weakly fair* scheduler must eventually select every *continuously* enabled process, while the *unfair* scheduler never need select an enabled process, unless it is the only enabled process.

We say that a process p is *neutralized* in the step $\gamma_i \mapsto \gamma_{i+1}$ if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action at that step. The neutralization of a process represents the following situation: at least one neighbor of p changes its state at the step $\gamma_i \mapsto \gamma_{i+1}$, and this change makes the guard of every action of p false.

To evaluate time complexity, we use the notion of *round*. The first *round* of an execution e is the minimum prefix of e in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. If γ_i is the last configuration of that round, the second round of e is defined to be the first round of the suffix e' of e which begins at γ_i, the third round of e is the second round of e', *etc.*

2.2 Self-Stabilization and Silence

Let $\mathcal{A}$ be a distributed algorithm and $\mathbb{P}$ be a predicate over the configurations of $\mathcal{A}$. $\mathcal{A}$ *stabilizes to* $\mathbb{P}$ if there exists a non-empty subset $\mathcal{S}$ of configurations of $\mathcal{A}$ such that:

Correction: $\forall \gamma \in \mathcal{S}$, $\mathbb{P}(\gamma)$.

Closure: For each possible step $\gamma \mapsto \gamma'$ of $\mathcal{A}$, $\gamma \in \mathcal{S} \Rightarrow \gamma' \in \mathcal{S}$.

Convergence: Each execution of $\mathcal{A}$ (starting from an arbitrary configuration) contains a configuration of $\mathcal{S}$.

The configurations of $\mathcal{S}$ are said to be *legitimate*, and other configurations are called *illegitimate*.

We say that an algorithm is *silent* [17] if each of its executions is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a configuration where none of its actions is enabled at any process. In this paper, we are interested in silent self-stabilizing algorithms. To show stabilization of such an algorithm $\mathcal{A}$ to a predicate $\mathbb{P}$, it is sufficient to show that (1) every execution of $\mathcal{A}$ is finite and (2) every terminal configuration of $\mathcal{A}$ satisfies $\mathbb{P}$.

3 Bound

In this section, we present an upper bound on the size of the minimum k-dominating set in any connected network. This upper bound originally appeared in [26]. However, the proof given in [26] overlooked a special case. The same case was overlooked in some other subsequent papers as well [25, 27].

Below, we exhibit a counterexample to show the special case where the proof of [26] is not valid. We then show how to fix the problem without affecting the upper bound.

Let T be an arbitrary spanning tree of $G = (V, E)$ rooted at some process r, that is, any connected graph $T = (V_T, E_T)$ such that $V_T = V$, $E_T \subseteq E$, and $|E_T| = |V_T| - 1$, where the process r is distinguished. In T, the *height* of process p, $h(p)$, denotes its distance to the root r. $H(T)$, the height of T, is defined to be $\max_{p \in V_T} h(p)$, and denoted simply H if T is understood. simply H when it is clear from the context. We write $H(T(p))$ for the height of the subtree $T(p)$ of T rooted at p.

The original proof consists of partitioning the processes of V into sets $T_0, \ldots, T_H$, where $T_i = \{p : h(p) = i\}$. These sets are merged into $k+1$ disjoint sets $D_0, \ldots, D_k$, where $D_i = \bigcup_{j \geq 0} T_{i+j(k+1)}$.

When $k < H$, the proof in [26] claims that (1) the size of the smallest set D_i is at most $\lceil \frac{n}{k+1} \rceil$, and (2) every D_i ($i \in [0..k]$) is k-dominating. The upper bound is then obtained by considering the set D_i of smallest size.

Actually, this latter set is not always k-dominating. The problem arises near the root, where there could be a process which has no ancestor in D_i. For example, consider the case $k = 2$ in the tree network of Figure 2. Clearly, D_2 is not a 2-dominating set, because u is not 2-dominated by any process in D_2; $\|u, w\| = 3$.

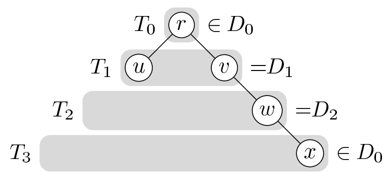

Figure 2: Counterexample of the original proof

This mistake can be corrected without changing the bound. Actually, the mistake only appears when the smallest D_i ($i \in [0..k]$), say D_ℓ, is not D_0. In this case, a leaf process whose height is strictly less than ℓ may be not k-dominated by any process in D_ℓ (as in the previous example). To correct this mistake we simply proceed as follows. When $k \geq H$ (in this case $\|D_0\| = 1$) or every D_i ($i \in [0..k]$) has the same size (*i.e.*, $\lceil \frac{n}{k+1} \rceil$), then we choose $D = D_0$. Otherwise, the size of the smallest D_i ($i \in [0..k]$), say $D_{\min}$, is strictly less than $\lceil \frac{n}{k+1} \rceil$ and we choose $D = D_{\min} \cup \{r\}$. In both cases, D is a k-dominating set of size at most $\lceil \frac{n}{k+1} \rceil$.

Theorem 1 *For every connected network $G = (V, E)$ of n processes and for every $k \geq 1$, there exists a k-dominating set D such that $|D| \leq \lceil \frac{n}{k+1} \rceil$.*

Proof. If $n = 0$, then $\lceil \frac{n}{k+1} \rceil = 0 = |\emptyset|$ and $\emptyset$ is a k-dominating set.

Assume now that $n > 0$. Let T be any rooted spanning tree of G rooted at some process r and let $D_0, \ldots, D_k$ be the $k + 1$ previously defined sets.

- Assume that $k \geq H$. Then, D_0 only contains r, and every other process is within distance k of r. So, D_0 is a k-dominating set of size $1 \leq \lceil \frac{n}{k+1} \rceil$.
- Assume that $k < H$. Then, for every $i \in [0..k]$, $|D_i| > 0$.

1. Assume that $\forall i \in [0..k-1]$, $|D_i| = |D_{i+1}|$. Then, $\forall i \in [0..k]$, $|D_i| = \lceil \frac{n}{k+1} \rceil$. Let $v \notin D_0$. The height of v, $h(v)$, satisfies $h(v) = x(k+1) + y$, where $x \geq 0$ and $0 < y \leq k$. Let u be the ancestor of v such that $h(u) = h(v) - y$ (such an ancestor exists because $y \leq h(v)$). By definition, $u \in D_0$ and $\|u, v\| \leq k$. Hence, D_0 is a k-dominating set such that $|D_0| = \lceil \frac{n}{k+1} \rceil$.
2. Assume that there exists $i \in [0..k-1]$ such that $|D_i| \neq |D_{i+1}|$. Let $\min \in [0..k]$ such that $\forall i \in [0..k]$, $|D_{\min}| \leq |D_i|$. Then, $|D_{\min}| < \lceil \frac{n}{k+1} \rceil$. Let $D = D_{\min} \cup \{r\}$. Then, $|D| \leq \lceil \frac{n}{k+1} \rceil$. Let $v \notin D$.
   (a) If $h(v) \leq k$, then v is at distance at most k from r and $r \in D$.
   (b) If $h(v) > k$, then $h(v) = x(k+1) + y$ with $x > 0$, $0 \leq y \leq k$, and $y \neq \min$. If $y > \min$, then let u be the ancestor of v such that $h(u) = x(k+1) + \min$. If $y < \min$, let u be the ancestor of v such that $h(u) = (x-1)(k+1) + \min$. By definition, $u \in D$ (more precisely, $u \in D_{\min}$) and $\|u, v\| \leq k$.

   Hence, D is a k-dominating set, and $|D| \leq \lceil \frac{n}{k+1} \rceil$.

□

4 Hierarchical Collateral Composition

Composition techniques are often used to simplify the design and the proofs of self-stabilizing algorithms [30]. A number of composition techniques have been given, among them, *collateral composition*, introduced by Herman [20], and *fair composition*, introduced by Dolev [16]. In collateral composition, the composition of two algorithms just consists of running the two algorithms concurrently, where the second algorithm uses the output of the first in its computations. When actions of both algorithms are enabled at the same process, the process nondeterministically executes one or the other. This nondeterminism is resolved in the fair composition by running the two algorithms in an alternating manner.

Here, we use a slightly modified version of the *collateral composition* [20], in which we resolve the nondeterminism of collateral composition as follows: when we compose two distributed algorithms $\mathcal{A}$ and $\mathcal{B}$, we modify the code of $\mathcal{B}(p)$ (for every process p) so that p executes an action of $\mathcal{B}(p)$ only when it has no enabled action in $\mathcal{A}(p)$.

Definition 1 (Hierarchical Collateral Composition) *Let $\mathcal{A}$ and $\mathcal{B}$ be two distributed algorithms on the same network, such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. In the* hierarchical collateral composition *of $\mathcal{A}$ and $\mathcal{B}$, noted $\mathcal{B} \circ \mathcal{A}$, the local program of every process p, $\mathcal{B}(p) \circ \mathcal{A}(p)$, is defined as follows:*

- *$\mathcal{B}(p) \circ \mathcal{A}(p)$ contains all variables of $\mathcal{A}(p)$ and $\mathcal{B}(p)$.*
- *$\mathcal{B}(p) \circ \mathcal{A}(p)$ contains all actions of $\mathcal{A}(p)$.*
- *For every action $G_i \rightarrow S_i$ of $\mathcal{B}(p)$, $\mathcal{B}(p) \circ \mathcal{A}(p)$ contains the action $\neg C_p \wedge G_i \rightarrow S_i$ where C_p is the disjunction of all guards of actions in $\mathcal{A}(p)$.*

Below, we give two properties of hierarchical collateral composition: Theorem 2 and Corollary 1. Corollary 1 gives a sufficient condition to show the correctness of the composite algorithm. To prove these properties, we need to first define the notions of *minimal relevant subsequence* and *projection.*

Definition 2 (MRS) *Let s be a sequence of configurations. The* minimal relevant subsequence *of s, noted $\mathcal{MRS}(s)$, is the maximal subsequence of s where no two consecutive configurations are identical.*

Definition 3 (Projection) *Let γ be a configuration and $\mathcal{A}$ be an algorithm. The* projection $\gamma_{|\mathcal{A}}$ *is the configuration obtained by removing from γ the values of all variables that do not exist in $\mathcal{A}$. If $e = \gamma_0 \ldots \gamma_i$ be a sequence of configurations, the* projection $e_{|\mathcal{A}}$ *is the sequence $\gamma_{0|\mathcal{A}} \ldots \gamma_{i|\mathcal{A}}$.*

Roughly speaking, the following theorem shows that if $\mathcal{A}$ is a silent self-stabilizing algorithm in the composite algorithm $\mathcal{B} \circ \mathcal{A}$, and the daemon is weakly fair, then $\mathcal{B}$ cannot prevent $\mathcal{A}$ from reaching a legitimate terminal configuration.

Theorem 2 *Let $\mathcal{A}$ be a silent algorithm that stabilizes to a predicate $\mathbb{P}_{\mathcal{A}}$ under a weakly fair daemon. Let $\mathcal{B}$ be an algorithm such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. $\mathcal{B} \circ \mathcal{A}$ satisfies the two following claims:*

1. *It stabilizes to $\mathbb{P}_{\mathcal{A}}$ under a weakly fair daemon.*

2. *It eventually reaches a configuration where no action of $\mathcal{A}$ is enabled ever.*

Proof. Let e be an execution of $\mathcal{B} \circ \mathcal{A}$ under the weakly fair daemon. Let $e' = \mathcal{MRS}(e_{|\mathcal{A}})$. No variable in the configurations of e' is written by $\mathcal{B}$, and all configurations of e' are possible configurations of $\mathcal{A}$.

Consider any process p continuously enabled *w.r.t.* algorithm $\mathcal{A}$ in a configuration γ of e'. Then, by construction, p is continuously enabled to execute an action of $\mathcal{A}$ from the first configuration of e that generates γ. Thus, it eventually executes an action of $\mathcal{A}$ in e and consequently in e'. Hence e' is a possible execution of $\mathcal{A}$ under the weakly fair daemon. Consequently, e' stabilizes to $\mathbb{P}_{\mathcal{A}}$ and is finite. Hence, e stabilizes to $\mathbb{P}_{\mathcal{A}}$ and eventually reaches a configuration where no action of $\mathcal{A}$ will ever be enabled again. □

From the previous theorem, we immediately deduce the following corollary:

Corollary 1 *$\mathcal{B} \circ \mathcal{A}$ stabilizes to a predicate $\mathbb{P}$ under the weakly fair daemon if the following conditions hold:*

1. *$\mathcal{A}$ is a silent self-stabilizing algorithm under the weakly fair daemon.*

2. *$\mathcal{B}$ stabilizes under the weakly fair daemon to $\mathbb{P}$ from any configuration where no action of $\mathcal{A}$ will ever be enabled again.*[1]

Proof. By Theorem 2.(2) and (1), any execution of $\mathcal{B} \circ \mathcal{A}$, assuming a weakly fair daemon, reaches a configuration γ from which no action of $\mathcal{A}$ is ever enabled. Then, from γ, $\mathcal{B}$ stabilizes to $\mathbb{P}$, by (2). □

5 Algorithm $\mathcal{SMDS}(k)$

In this section, we present a silent self-stabilizing algorithm, called $\mathcal{SMDS}(k)$ (*Small Minimal k-Dominating Set*), which builds a minimal k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes in any identified network, assuming a weakly fair daemon. This algorithm is a hierarchical collateral composition of three silent self-stabilizing algorithms, $\mathcal{SMDS}(k) = \mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST}$, where:

- $\mathcal{ST}$ builds a rooted spanning tree.

- $\mathcal{DS}(k)$ computes a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes, using the spanning tree built by $\mathcal{ST}$.

- $\mathcal{MIN}(k)$ reduces the k-dominating set built by $\mathcal{DS}(k)$ to a minimal k-dominating set by deleting processes.

We give more details about the three layers of $\mathcal{SMDS}(k)$ in Subsections 5.1 to 5.3. The complexity of $\mathcal{SMDS}(k)$ is proven in Subsection 5.4.

[1] Recall that in such a configuration, the specification of $\mathcal{A}$ is satisfied.

5.1 Algorithm $\mathcal{ST}$

$\mathcal{ST}$ is any silent self-stabilizing spanning tree algorithm for arbitrary identified networks which works under the weakly fair daemon. The spanning tree built by $\mathcal{ST}$ is *rooted*, meaning that some process of the tree is distinguished as the *root* For each other process p, let the *parent* of p, be the neighbor of p on the unique shortest path, through the tree, from p to the root. We assume that the output of $\mathcal{ST}$ is the macro $\texttt{Parent}_p$, which is defined for all processes p. $\texttt{Parent}_p$ returns $\perp$ if p believes to be the root of the spanning tree, otherwise $\texttt{Parent}_p$ returns the parent of p. $\mathcal{ST}$ stabilizes to the predicate $SP_{\mathcal{ST}}$ defined as follows: $SP_{\mathcal{ST}}$ holds if and only if the configuration is terminal, there exists a unique process r such that $\texttt{Parent}_r = \perp$, and the graph $T = (V, E_T)$ where $E_T = \{\{p, \texttt{Parent}_p\}, \forall p \in V \setminus \{r\}\}$ is a spanning tree.

The silent self-stabilizing algorithm for identified networks given in [12] can be used to implement $\mathcal{ST}$. Actually, this algorithm elects a leader; however, as with most existing silent self-stabilizing leader election algorithms, it also builds a spanning tree that is rooted at the leader process. This algorithm stabilizes in $O(n)$ rounds using $O(\log n)$ bits per process, and does not require processes to know any upper bound on the size n or the diameter $\mathcal{D}$ of the network.

From [12], we have:

Lemma 1 *$\mathcal{ST}$ is a silent algorithm which stabilizes to $SP_{\mathcal{ST}}$ under a weakly fair daemon.*

5.2 Algorithm $\mathcal{DS}(k)$

$\mathcal{DS}(k)$ (see Algorithm 1 for the formal description) is also silent and uses the spanning tree T built by $\mathcal{ST}$ to compute a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. It is based on the construction given in the proof of Theorem 1 (page 120). Informally, $\mathcal{DS}(k)$ uses the following three variables at each process p:

- $p.color \in [0..k]$. Using this variable, p computes $h(p) \bmod (k+1)$ (that is its height in T modulo $k+1$) in top-down fashion using Action FixColor. Hence, once $\mathcal{DS}(k)$ has stabilized, each set D_i, defined in Section 3, corresponds to the set $\{p \in V \mid p.color = i\}$.

- *The integer array $p.pop[i]$ is defined for all $i \in [0..k]$.* In each cell $p.pop[i]$, p computes the number of processes in its subtree $T(p)$ having color i, that is, processes q such that $q.color = i$. This computation is performed in a bottom-up fashion using Action FixPop. Hence, once $\mathcal{DS}(k)$ has stabilized, r knows the size of each set D_i.

- $p.min \in [0..k]$. In this variable, p computes the smallest index of the smallest non-empty set D_i, that is, the least used value to color some processes of the network. This value is evaluated in a top-down fashion using Action FixMin based on the values computed in the array $r.pop$. Once the values of $r.pop$ are correct, the root r can compute in $r.min$ the least used color (in case of equality, the smallest index is chosen). Then, the value of $r.min$ is broadcast down in the tree.

According to Theorem 1 (page 120), after $\mathcal{DS}(k)$ has stabilized, the set of processes p such that $p = r$ or $p.color = p.min$, *i.e.*, the set $\{p \in V \mid IsDominator_p\}$, is a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. So, $\mathcal{DS}(k) \circ \mathcal{ST}$ stabilizes to the predicate $SP_{\mathcal{DS}(k)}$ defined as follows: $SP_{\mathcal{DS}(k)}$ holds if and only if the configuration is terminal and the set $\{p \in V \mid IsDominator_p = \text{TRUE}\}$ is a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes.

We now show the correctness of $\mathcal{DS}(k)$. In the following proofs, we always assume the system starts from a configuration where no action of $\mathcal{ST}$ is enabled. Since $\mathcal{DS}(k)$ does not write into the variables of $\mathcal{ST}$, all variables of $\mathcal{ST}$ are fixed forever in such a configuration. Moreover, a spanning tree is well-defined (using the input $\texttt{Parent}_p$ of every process p) by Lemma 1. We denote this spanning tree by T and its root by r.

Lemma 2 *Starting from any configuration where no action of $\mathcal{ST}$ is enabled, the variable $p.color$ of every process p is set forever to $h(p) \bmod (k+1)$ in at most n rounds.*

Proof. First, remark that:

(a) For every process p, Action FixColor, whose guard is $\neg ColorOK_p$, is the only action of p that modifies $p.color$.

We show the lemma by induction on the height of the processes in T.
Let γ be a configuration where no action of $\mathcal{ST}$ is enabled.

- **Base Case:** Let consider the root r (the only process of height 0).

  (b) Predicate $ColorOK_r$ only depends on the variable $r.color$ and input the $\texttt{Parent}_r$, which is set forever to $\perp$ at γ.

  Assume that $ColorOK_r$ holds in γ. Then, $r.color = 0$. Moreover, by (a) and (b), $ColorOK_r$ holds forever and, consequently, $r.color = 0$ holds forever.

  Assume that $ColorOK_r$ does not hold in γ. Then, by (a) and (b), Action FixColor is continuously enabled at r. As the daemon is weakly fair, Action FixColor is executed by r in at most one round. Hence, after at most one round from γ, $ColorOK_r$ becomes true and we reduce to the previous case.

- **Inductive Hypothesis:** Let $j \in \mathbb{N}^*$. Assume that, for every process p such that $h(p) < j$, the variable $p.color$ is set forever to $h(p) \bmod (k+1)$ after at most $h(p) + 1$ rounds from γ.

- **Inductive Step:** Consider any process p such that $h(p) = j$.

  (c) The predicate $ColorOK_p$ depends only on the variable $p.color$, input $\texttt{Parent}_p$ which is fixed to some value in $\mathcal{N}_p$ from γ, and $\texttt{Parent}_p.color$ which is set forever to $h(\texttt{Parent}_p) \bmod (k+1)$ after at most $h(p)$ rounds from γ by the inductive hypothesis.

  Assume that $ColorOK_p$ holds after $h(p)$ rounds from γ. Then, $p.color = (\texttt{Parent}_p.color + 1) \bmod (k+1) = (h(\texttt{Parent}_p) \bmod (k+1) + 1) \bmod (k+1) = h(p) \bmod (k+1)$. Moreover, by (a) and (c), $ColorOK_p$ holds forever and, consequently, $p.color = h(p) \bmod (k+1)$ holds forever.

  Assume that $ColorOK_p$ does not hold after $h(p)$ rounds from γ. Then by (a) and (c), Action FixColor is continuously enabled at p from γ. As the daemon is weakly fair, Action FixColor is executed by p in at most one additional round. Hence, in at most $h(p) + 1$ rounds from γ, $ColorOK_p$, becomes true and we reduce to the previous case.

As the height of T is bounded by $n - 1$, the lemma holds. □

Lemma 3 *Starting from any configuration where:*

- *no action of $\mathcal{ST}$ is enabled, and*
- *the variable $q.color$ of every process q is set forever to $h(q) \bmod (k+1)$,*

for every process p and every index $i \in [0..k]$, the variable $p.pop[i]$ is set forever to $|\{q \in T(p) \mid q.color = i\}|$ in at most n rounds.

Proof. First, we remark that:

(a) For every process p, Action FixPop, whose guard is $ColorOK_p \wedge \neg PopOK_p$, is the only action of p that modifies $p.pop$.

Let γ be a configuration where:

- no action of $\mathcal{ST}$ is enabled, and
- the variable $q.color$ of every process q is set forever to $h(q) \bmod (k+1)$.

We remark that:

(b) From γ, for every process p, $ColorOK_p$ holds forever and, consequently, Action FixPop is enabled at p if and only if $\neg PopOK_p$ holds.

We now show the lemma by induction on the height of $T(p)$.

- **Base Case:** Consider any process p such that $H(T(p)) = 0$ (p is a leaf process).

  (c) Predicate $PopOK_p$ depends only on the variables $p.pop$ and $p.color$, the latter being set forever to $h(p) \bmod (k+1)$ starting from γ.

  Assume that $PopOK_p$ holds in γ. Then, $\forall i \in [0..k]$, $p.pop[i] = \mathtt{SelfPop}_p(i) = |\{q \in T(p) \mid q.color = i\}|$. Moreover, by (a)-(c), $PopOK_p$ holds forever and, and consequently, $\forall i \in [0..k]$, $p.pop[i] = |\{q \in T(p) \mid q.color = i\}|$ holds forever.

  Assume that $PopOK_p$ does not hold in γ. Then by (a)-(c), Action FixPop is continuously enabled. As the daemon is weakly fair, Action FixPop is executed by p in at most one round from γ. Then, $PopOK_p$ becomes true, and we reduce to the previous case.

- **Inductive Hypothesis:** Let $j \in \mathbb{N}^*$. Assume that for every process p such that $H(T(p)) < j$ and every index $i \in [0..k]$, the variable $p.pop[i]$ is set to $|\{q \in T(p) \mid q.color = i\}|$ after at most $H(T(p)) + 1$ rounds from γ.

- **Inductive Step:** Consider any process p such that $H(T(p)) = j$.

  (d) The predicate $PopOK_p$ depends only on variables $p.pop$, $p.color$ (which is fixed by assumption), and $q.pop$ of every child q of p in T; these latter variables are fixed after $H(T(p))$ rounds from γ, by the inductive hypothesis.

  Assume that $PopOK_p$ holds after $H(T(p))$ rounds from γ. Then, $\forall i \in [0..k], p.pop[i] = \mathtt{EvalPop}_p(i)$, *i.e.*, $p.pop[i] = \mathtt{SelfPop}_p(i) + \sum_{q \in \mathtt{Children}_p} |\{q' \in T(q) \mid q'.color = i\}| = |\{q \in T(p) \mid q.color = i\}|$, by the inductive hypothesis. Moreover, by (a), (b), and (d), $PopOK_p$ holds forever and, consequently, $\forall i \in [0..k], p.pop[i] = |\{q \in T(p) \mid q.color = i\}|$ holds forever.

  Assume that $PopOK_p$ does not hold after $H(T(p))$ rounds from γ. Then, by (a), (b), and (d), Action FixPop is continuously enabled at p. As the daemon is assumed to be weakly fair, p executes Action FixPop in at most 1 round. Hence, in at most $H(T(p)) + 1$ rounds, $PopOK_p$ becomes true, and we reduce to the previous case.

As the height of T is bounded by $n - 1$, the lemma holds. □

The proof of the next lemma follows the same scheme as that of Lemma 2.

Lemma 4 *Starting from any configuration where:*

- *no action of $\mathcal{ST}$ is enabled,*
- *the variable $p.color$ of every process p is set forever to $h(p) \bmod (k+1)$, and*
- *for every process p and every index $i \in [0..k]$, the variable $p.pop[i]$ is set forever to $|\{q \in T(p) \mid p.color = i\}|$,*

in at most n rounds, the variable $p.min$ of every process p is set forever to the smallest index $i_{min} \in [0..k]$ that satisfies $|C_{i_{min}}| = \min_{j \in [0..k] \ \mid\ C_j \neq \emptyset} |C_j|$, where $C_j = \{q \in T \mid q.color = j\}$ for every $j \in [0..k]$.

From Lemmas 2 to 4, we obtain the following theorem:

Theorem 3 *Starting from any configuration where no action of $\mathcal{ST}$ is enabled, $\mathcal{DS}(k) \circ \mathcal{ST}$ converges in at most $3n$ rounds to a terminal configuration where, for every process p:*

(a) $p.color = h(p) \bmod (k+1)$, *and*

(b) $p.min = i_{min}$ *where* i_{min} *is the smallest index in* $[0..k]$ *that satisfies* $|C_{i_{min}}| = \min_{j\in[0..k] \mid C_j\neq\emptyset} |C_j|$, *where* $C_j = \{q \in T \mid q.color = j\}$ *for every* $j \in [0..k]$.

We now consider any terminal configuration γ_t of $\mathcal{DS}(k) \circ \mathcal{ST}$ (such a configuration exists by Corollary 1, Lemma 1 and Theorem 3). Let c_t be the unique value in the variables $\{p.min\}$ in γ_t (c_t is well-defined by Theorem 3). In γ_t, the output of $\mathcal{DS}(k) \circ \mathcal{ST}$ is the set $DS^{out} = \{p \in V \mid IsDominator_p\}$.

From Theorem 3 and definition of predicate $IsDominator_p$, we can deduce the following lemma:

Lemma 5 *In* γ_t, $DS^{out} = \{r\} \cup DS^{c_t}$ *where* $DS^{c_t} = \{p \in V \mid h(p) \bmod (k+1) = c_t\}$.

We now show that, in any case, DS^{out} is the same set as the one obtained by applying the constructive method given in the proof of Theorem 1.

We first recall some definitions. We divide the processes into sets $T_0, \ldots, T_H$ according to their height in the tree, and assign all the processes of height i to T_i. These sets are merged into $k+1$ sets $D_0, \ldots, D_k$ by taking $D_i = \bigcup_{j\geq 0} T_{i+j(k+1)}$.

Remark 1 $DS^{c_t} = D_{c_t}$.

Theorem 4 *In* γ_t, DS^{out} *is a k-dominating set of G, and* $|DS^{out}| \leq \lceil \frac{n}{k+1} \rceil$.

Proof. We have three cases.

- $k \geq H$. In this case, the proof of Theorem 1 states that D_0 is a k-dominating set of size at most $\lceil \frac{n}{k+1} \rceil$. By Theorem 3.(b), c_t is the smallest index in $[0..k]$ such that $|C_{c_t}| = \min_{j\in[0..k] \mid C_j\neq\emptyset} |C_j|$, where $C_j = \{q \in T \mid q.color = j\}$ for every $j \in [0..k]$. Moreover, by Theorem 3.(a), $C_j = D_j$ for every $j \in [0..k]$. Thus, c_t is the smallest index in $[0..k]$ such that $|D_{c_t}| = \min_{j\in[0..k] \mid D_j\neq\emptyset} |D_j|$. By definition, $\min_{j\in[0..k] \mid D_j\neq\emptyset} |D_j| \geq 1$. Now, as $k \geq H$, $D_0 = \{r\}$, *i.e.*, $|D_0| = 1$ and $c_t = 0$. Hence, $DS^{c_t} = D_0$ by Remark 1, and $DS^{out} = \{r\} \cup D_0 = D_0$, and we are done.

- $k < H$ *and for every* $i \in [0..k-1]$, $|D_i| = |D_{i+1}|$. The proof is similar to that of the previous case.

- $k < H$ *and there exists* $i \in [0..k-1]$ *such that* $|D_i| \neq |D_{i+1}|$. Let i_{min} the smallest index such that $|D_{i_{min}}| = \min_{j\in[0..k] \mid D_j\neq\emptyset} |D_j|$. In this case, the proof of Theorem 1 states that $\{r\} \cup D_{i_{min}}$ is a k-dominating set of size at most $\lceil \frac{n}{k+1} \rceil$. By Theorem 3.(b), c_t is the smallest index in $[0..k]$ that satisfies $|C_{c_t}| = \min_{j\in[0..k] \mid C_j\neq\emptyset} |C_j|$ where $C_j = \{q \in T \mid q.color = j\}$ for every $j \in [0..k]$.

  Moreover, by Theorem 3.(a), $C_j = D_j$ for every $j \in [0..k]$. Thus, c_t is the smallest index in $[0..k]$ such that $|D_{c_t}| = \min_{j\in[0..k] \mid D_j\neq\emptyset} |D_j|$. Hence, $c_t = i_{min}$, $DS^{c_t} = D_{i_{min}}$ by Remark 1, $DS^{out} = \{r\} \cup D_{i_{min}}$, and we are done.

In all cases, DS^{out} is equal to the set obtained by applying the constructive method given in the proof of Theorem 1. Hence, the theorem holds. □

From Theorems 3 and 4, we can deduce the following theorem:

Theorem 5 *Starting from any configuration where no action of $\mathcal{ST}$ is enabled, algorithm $\mathcal{DS}(k)$ converges, in at most $3n$ rounds to a terminal configuration satisfying* $SP_{\mathcal{DS}(k)}$.

From Corollary 1, Lemma 1 and Theorem 5, we can deduce the following theorem:

Theorem 6 $\mathcal{DS}(k) \circ \mathcal{ST}$ *is silent and stabilizes to* $SP_{\mathcal{DS}(k)}$ *within* $O(n)$ *rounds, under a weakly fair daemon.*

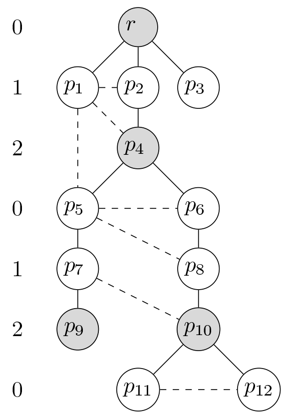

Figure 3: Example of 2-dominating set computed by our algorithm

Figure 3 shows an example of a 2-dominating set computed by $\mathcal{DS}(2) \circ \mathcal{ST}$. In the figure, bold lines represent tree-edges, and dashed lines indicate non-tree-edges. In this example, $r.pop[0] = 5$, $r.pop[1] = 5$ and $r.pop[2] = 3$ once $\mathcal{DS}(2) \circ \mathcal{ST}$ stabilizes. Thus, $r.min = 2$, which means that the smallest color in use is 2. $D_2 = \{p_4, p_9, p_{10}\}$ and $|D_2| = 3$. In this case, the 2-dominating set that $\mathcal{DS}(2) \circ \mathcal{ST}$ eventually outputs is $SD = \{r\} \cup D_2$, *i.e.*, $\{r, p_4, p_9, p_{10}\}$. This 2-dominating set follows the bound given in Theorem 1 (page 120), as the size of SD is 4, which is less than $\lceil \frac{13}{2+1} \rceil = 5$. However, SD is not minimal. For example, $\{r, p_{10}\}$ is a proper subset of SD that is 2-dominating, and is in fact minimal.

5.3 Algorithm $\mathcal{MIN}(k)$

$\mathcal{MIN}(k)$ is also silent and computes a minimal k-dominating set which is a subset of the k-dominating set computed by $\mathcal{DS}(k)$. In Section 7, we will see that the minimization performed by $\mathcal{MIN}(k)$ provides an improvement which is not negligible.

This last layer of our algorithm can be achieved using the silent self-stabilizing algorithm $\mathcal{MIN}(k)$ given in [10]. This algorithm takes a k-dominating set I as input, and constructs a subset of I that is a minimal k-dominating set. The knowledge of I is distributed meaning that every process p uses only the input $IsDominator_p$ to know whether it is in the k-dominating set or not. Based on this input, $\mathcal{MIN}(k)$ assigns the output Boolean variable $p.inD$ of every process p in such way that eventually $\{p \in V \mid p.inD = \text{TRUE}\}$ is a minimal k-dominating set of the network.

Using the output of algorithm $\mathcal{DS}(k) \circ \mathcal{ST}$ as input for algorithm $\mathcal{MIN}(k)$, the size of the resulting minimal k-dominating set remains bounded by $\lceil \frac{n}{k+1} \rceil$, because $\mathcal{MIN}(k)$ can only remove processes in the k-dominating set computed by $\mathcal{DS}(k)$. Hence, $\mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST}$ stabilizes to the predicate $SP_{\mathcal{SMDS}(k)}$, which is the conjunction of the following two conditions:

1. The configuration is terminal.
2. The set $\{p \in V \mid p.inD = \text{TRUE}\}$ is a minimal k-dominating size at most $\lceil \frac{n}{k+1} \rceil$.

As $\mathcal{SMDS}(k) = \mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST}$, from Corollary 1 and Theorem 6, we have:

Theorem 7 (Overall Correctness) $\mathcal{SMDS}(k)$ *is silent, and stabilizes to* $SP_{\mathcal{SMDS}(k)}$ *under a weakly fair daemon.*

5.4 Complexity Analysis

We first consider the round complexity of $\mathcal{SMDS}(k)$. Using the algorithm of [12], $\mathcal{ST}$ stabilizes in $O(n)$ rounds. Once the spanning tree is available, $\mathcal{DS}(k)$ stabilizes in $O(n)$ rounds, by Theorem 6.

Algorithm 1 $\mathcal{DS}(k)$, code for each process p

Inputs:

$\texttt{Parent}_p \in \mathcal{N}_p \cup \{\perp\}$	Parent process of p in the spanning tree, $\perp$ for the root.

Variables:

$p.color \in [0..k]$	Color of p.
$p.pop[i]$ for all $i \in [0..k]$	Population, *i.e.,* number of processes, of color i in $T(p)$.
$p.min \in [0..k]$	Color with the smallest population in $T(p)$.

Macros:

$\texttt{EvalColor}_p$	$=$	0 **if** $(\texttt{Parent}_p = \perp)$ **else** $(\texttt{Parent}_p.color + 1) \bmod (k+1)$
$\texttt{SelfPop}_p(i)$	$=$	1 **if** $(p.color = i)$ **else** 0
$\texttt{Children}_p$	$=$	$\{q \in \mathcal{N}_p \mid \texttt{Parent}_q = p\}$
$\texttt{EvalPop}_p(i)$	$=$	$\texttt{SelfPop}_p(i) + \sum_{q \in \texttt{Children}_p} q.pop[i]$
$\texttt{MinPop}_p$	$=$	$\min_{i\in[0..k]} \{p.pop[i] \mid p.pop[i] > 0\}$
$\texttt{MinColor}_p$	$=$	$\min_{i\in[0..k]} \{i \mid p.pop[i] = \texttt{MinPop}_p\}$
$\texttt{EvalMin}_p$	$=$	$\texttt{MinColor}_p$ **if** $(\texttt{Parent}_p = \perp)$ **else** $\texttt{Parent}_p.min$

Predicates:

$IsRoot_p$	$\equiv$	$\texttt{Parent}_p = \perp$
$ColorOK_p$	$\equiv$	$p.color = \texttt{EvalColor}_p$
$PopOK_p$	$\equiv$	$\forall i \in [0..k], p.pop[i] = \texttt{EvalPop}_p(i)$
$MinOK_p$	$\equiv$	$p.min = \texttt{EvalMin}_p$
$IsDominator_p$	$\equiv$	$IsRoot_p \vee (p.color = p.min)$

Actions:

FixColor	::	$\neg ColorOK_p$	$\longrightarrow$	$p.color \leftarrow \texttt{EvalColor}_p$
FixPop	::	$ColorOK_p \wedge \neg PopOK_p$	$\longrightarrow$	$\forall i \in [0..k], p.pop[i] \leftarrow \texttt{EvalPop}_p(i)$
FixMin	::	$ColorOK_p \wedge PopOK_p \wedge \neg MinOK_p$	$\longrightarrow$	$p.min \leftarrow \texttt{EvalMin}_p$

Finally, the k-dominating set computed by the first two layers is minimized by $\mathcal{MIN}(k)$ in $O(n)$ rounds (see [10]). Thus, we have:

Theorem 8 *$\mathcal{SMDS}(k)$ stabilizes to $SP_{\mathcal{SMDS}(k)}$ in $O(n)$ rounds.*

We now consider the space complexity of $\mathcal{SMDS}(k)$. $\mathcal{ST}$ and $\mathcal{MIN}(k)$ can be implemented using $O(\log n)$ bits per process [12, 10]. $\mathcal{DS}(k)$ at each process is composed of two variables whose domain has $k+1$ elements, and an array of $k+1$ integers. However, in the terminal configuration, the minimum non-null value of a cell is at most $\lceil \frac{n}{k+1} \rceil$. Thus, the algorithm still works if we replace any assignment of any value val to a cell by $\min(val, \lceil \frac{N}{k+1} \rceil + 1)$, where N is any upper bound on n. In this case, each array can be implemented using $O(k \log \frac{N}{k})$ bits. Note that this bound can be obtained only if we assume that each process knows the upper bound N. However, n can be computed dynamically using the spanning tree.

Theorem 9 *$\mathcal{SMDS}(k)$ can be implemented using $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, where N is any upper bound on n.*

6 The Transformer

In the previous section, we show that $\mathcal{SMDS}(k)$ stabilizes to $SP_{\mathcal{SMDS}(k)}$ under a weakly fair daemon. We now give an automatic method for transforming any self-stabilizing algorithm which works under a weakly fair daemon into a self-stabilizing algorithm which works under an unfair daemon (for the same specification). Our method preserves the silence property of the input algorithm.

There already exist several methods to transform a weakly fair algorithm into an unfair one. In [2], the authors define the *cross-over* composition. Using this composition, a weakly fair algorithm can be transformed by composing it with an algorithm that is fair under an unfair daemon.[2] However, this technique does not preserve the silence of the input algorithm. Moreover, no step complexity analysis is given for the output unfair algorithm. In [24], authors give a transformer that preserves the silence of the input algorithm. Furthermore, the step complexity the transformed algorithm is $O(n^4 \times R)$, where R is the stabilization time of the input algorithm in rounds. Finally, note that the round complexity of the transformed version is much higher than that of the input algorithm (of the same order as the step complexity).

In contrast with the previous solutions, our transformer does not degrade the round complexity of the algorithm. Moreover, the step complexity of the transformed algorithm is $O(\mathcal{D}n(R + n^2))$, where R is the stabilization time of the input algorithm in rounds.

Let $\mathcal{A}$ be an algorithm that stabilizes to $SP_{\mathcal{A}}$, assuming a weakly fair daemon.[3] Let p be a process. We recall that $\mathcal{A}(p)$ denotes the local program of p in $\mathcal{A}$. Assume that $\mathcal{A}(p)$ has x actions. Actions of $\mathcal{A}(p)$ are indexed by $[0..x-1]$, and are of the following form:

$$A_i \ :: \ G_i \ \longrightarrow \ S_i.$$

We denote by $\mathcal{A}^t$ the transformed version of $\mathcal{A}$. $\mathcal{A}^t$ is obtained by composing $\mathcal{A}$ with a self-stabilizing phase clock algorithm. This latter algorithm, called $\mathcal{U}$, is treated as a black box ($\mathcal{U}(p)$ denotes the local program of p in $\mathcal{U}$), with the following properties:

1. Every process p has an incrementing variable $p.clock \in \mathbb{Z}_\alpha$, the cyclic group of order α, where $\alpha \geq 3$

2. The phase clock is self-stabilizing, assuming an unfair daemon, *i.e.*, after it has stabilized, there exists an integer function f on processes such that:

   - $f(p) \bmod \alpha = p.clock$
   - For all processes p and q, $|f(p) - f(q)| \leq \|p, q\|$.
   - For every process p, $f(p)$ increases by 1 infinitely often using statement $Incr_p$.

3. Every process p has in its local program $\mathcal{U}(p)$ an action $I :: Can\_Incr_p \rightarrow Incr_p$ such that, once $\mathcal{U}$ is stabilized, I is the only action that p is enabled to execute. Moreover, $\mathcal{U}$ does not require execution of Action I during the stabilization phase.

An algorithm that matches all these requirements can be found in [5].

The local program of each process p in $\mathcal{A}^t$ is obtained as follows:

- $\mathcal{A}^t(p)$ contains all variables of $\mathcal{A}(p)$ and $\mathcal{U}(p)$.

- $\mathcal{A}^t(p)$ contains all actions of $\mathcal{U}(p)$, except I, which is replaced by the following actions:

  - $A'_i :: Can\_Incr_p \wedge G_i \rightarrow Incr_p, S_i$ for every $i \in [0..x-1]$,
  - $L :: Can\_Incr_p \wedge Stable_p \wedge Late_p \rightarrow Incr_p$ where $Stable_p \equiv (\forall i \in [0..x-1] \mid \neg G_i)$ and $Late_p \equiv \neg(\forall q \in \mathcal{N}_p \mid q.clock = p.clock)$

[2] *I.e.*, an algorithm which guarantees that every process executes an infinite number of steps under an unfair daemon.

[3] In particular, if $\mathcal{A}$ is silent, any configuration of $\mathcal{A}$ satisfying $SP_{\mathcal{A}}$ is terminal.

Roughly speaking, our transformer enforces fairness among processes that are enabled in $\mathcal{A}$ because they can only move once at each clock tick. Once $\mathcal{A}$ has stabilized, if $\mathcal{A}$ is silent, then every process p eventually satisfies $Stable_p$ and, once all clocks have the same value, no further action is enabled, hence the silence is preserved.

Theorem 10 *$\mathcal{A}^t$ stabilizes to $SP_{\mathcal{A}}$ under an unfair daemon.*

Proof. By construction, any execution of $\mathcal{A}^t$ converges to a configuration γ' that is legitimate *w.r.t.* algorithm $\mathcal{U}$. Consider any configuration γ'' reachable from γ'. Assume that $\bigvee_{i\in[0..x-1]} G_i$ continuously holds at process p from γ'' but p never again executes any A'_i. $Stable_p$ is false forever from γ'' and, consequently, $p.clock$ is never again incremented. As $\mathcal{U}$ works under an unfair daemon, eventually every process $q \neq p$ is disabled. In this case, $f(p)$ is minimum in the system. In particular, Can_Incr_p holds. Thus, p is enabled to execute some A'_i. Hence, p is the only enabled process and it executes one of its enabled actions A'_i in the next step. Thus, if $\bigvee_{i\in[0..x-1]} G_i$ continuously holds at p from γ'', then p eventually executes one of its enabled actions A'_i in $\mathcal{A}^t$. As $\mathcal{A}$ stabilizes under a weakly fair daemon, $\mathcal{A}^t$ stabilizes to the same specification under an unfair daemon. □

Theorem 11 *If $\mathcal{A}$ is silent, then $\mathcal{A}^t$ is silent.*

Proof. First, by Theorem 10 (and its proof), $\mathcal{A}^t$ converges to a configuration γ from which both the specification of algorithm $\mathcal{U}$ and the predicate $Stable_p$ for every process p hold forever. So, from γ, only Action L can be executed by processes. Let $M = \max_{p\in V} f(p)$, and $m = \min_{p\in V} f(p)$. While $M \neq m$, only processes q such that $f(q) \neq M$ could be enabled to execute Action L. Moreover, when executing Action L, any q increases $f(q)$ by 1. Hence, eventually, $M = m$ and no action is ever again enabled in the system. □

Below, we present the complexity of the transformed algorithm. These results assume that $\mathcal{U}$ is the algorithm of Boulinier *et al.* in [5] The authors show that $2n-1$ states per process (actually the range of the phase clock) are sufficient to make $\mathcal{U}$ work in any topology (the worst case being the cycle topology). Moreover, using $2n-1$ states, the stabilization time of $\mathcal{U}$ is in $O(n)$ rounds [4] and $O(\mathcal{D}n^3)$ steps [14], respectively. Hence, we have the following theorem:

Theorem 12 *The space complexity of $\mathcal{A}^t$ is $O(\log n) + MEM$ bits per process, where MEM is the memory requirement for $\mathcal{A}$.*

Below, we prove an additional result about $\mathcal{U}$:

Lemma 6 *Once $\mathcal{U}$ is stabilized, every process advances its local clock of $\mathcal{D}$ ticks at most every $2\mathcal{D}$ rounds.*

Proof. Let $f_\gamma^{\min} = \min_{p\in V} f(p)$ in some configuration γ after $\mathcal{U}$ stabilized. Let q be a process and f_γ^q be the value of $f(q)$ in γ. $f_\gamma^{\min} \leq f_\gamma^q \leq f_\gamma^{\min} + \mathcal{D}$. $2\mathcal{D}$ rounds after γ, $f(q) \geq f_\gamma^{\min} + 2\mathcal{D}$. Thus, $f(q) - f_\gamma^q \geq f_\gamma^{\min} + 2\mathcal{D} - (f_\gamma^{\min} + \mathcal{D})$, *i.e.*, $f(q) - f_\gamma^q \geq \mathcal{D}$. That is, q increments its phase clock at least $\mathcal{D}$ times during that period. □

Theorem 13 *$\mathcal{A}^t$ stabilizes to $SP_{\mathcal{A}}$ in $O(n + \lceil \frac{R}{\mathcal{D}} \rceil \times 2\mathcal{D})$ rounds, where R is the stabilization time of $\mathcal{A}$ in rounds, and if $\mathcal{A}$ is silent, then $\mathcal{A}^t$ reaches a terminal configuration in a round complexity of the same order of magnitude.*

Proof. First, $\mathcal{A}^t$ stabilizes to the specification of $\mathcal{U}$ in $O(n)$ rounds. Then, $\mathcal{A}^t$ needs to emulate at most R rounds of $\mathcal{A}$ to stabilize to $SP_{\mathcal{A}}$. By Lemma 6, this requires at most $\lceil \frac{R}{\mathcal{D}} \rceil \times 2\mathcal{D}$ rounds.

Assume that $\mathcal{A}$ is silent. Then, consider the first configuration γ of $\mathcal{A}^t$ that is legitimate *w.r.t.* $SP_{\mathcal{A}}$ and the specification of $\mathcal{U}$. Let $M = \max_{p\in V} f(p)$, and $m = \min_{p\in V} f(p)$ in γ. Then, $M - m \leq \mathcal{D}$. Hence, by Lemma 6, after at most $2\mathcal{D}$ additional rounds, $\mathcal{A}^t$ reaches a terminal configuration, and we are done. □

The next lemma gives a bound on the number of steps required to emulate a round of $\mathcal{A}$, once $\mathcal{U}$ has stabilized.

Lemma 7 *Once $\mathcal{U}$ has stabilized, every continuously enabled process in $\mathcal{A}^t$ executes an action after at most $2\mathcal{D}(n-1)$ steps.*

Proof. Consider a configuration γ after $\mathcal{U}$ has stabilized, and a process p that is continuously enabled from γ.

Then, $f(p) - \|p,q\| \leq f(q) \leq f(p) + \|p,q\|$ for every process $q \neq p$. So, every process $q \neq p$ can increment $q.clock$ at most $2\|p,q\|$ times before $p.clock$ is incremented. So, at most $\sum_{q \in V \setminus \{p\}} 2\|p,q\|$ steps can occur before p executes an action. As $\sum_{q \in V \setminus \{p\}} 2\|p,q\| \leq (n-1) \times 2\mathcal{D}$, the lemma holds. $\square$

Theorem 14 *$\mathcal{A}^t$ stabilizes to $SP_{\mathcal{A}}$ in $O(\mathcal{D}n(R+n^2))$ steps, where R is the stabilization time of $\mathcal{A}$ in rounds; and if $\mathcal{A}$ is silent, then $\mathcal{A}^t$ reaches a terminal configuration, and its step complexity has the same order of magnitude.*

Proof. First, $\mathcal{A}^t$ stabilizes the specification of algorithm $\mathcal{U}$ in $O(\mathcal{D}n^3)$ steps. Then, by Lemma 7, we have that R rounds of $\mathcal{A}$ are emulated by $\mathcal{A}^t$ in $O(\mathcal{D}nR)$ steps.

Assume that $\mathcal{A}$ is silent. Then, consider the first configuration γ of $\mathcal{A}^t$ that is legitimate *w.r.t.* $SP_{\mathcal{A}}$ and the specification of $\mathcal{U}$. Let $M = \max_{p \in V} f(p)$, and $m = \min_{p \in V} f(p)$ in γ. Then, $M - m \leq \mathcal{D}$. Hence, after $O(\mathcal{D}n)$ additional steps, $\mathcal{A}^t$ reaches a terminal configuration, and we are done. $\square$

As a case study, $\mathcal{SMDS}(k)^t$ stabilizes to $SP_{\mathcal{SMDS}(k)}$ in $O(n)$ rounds and $O(\mathcal{D}n^3)$ steps using $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, by Theorems 8-9 and 12-14. This shows that our transformer does not degrade the round complexity and memory requirements while achieving an interesting step complexity.

7 Simulations

7.1 Model and Assumptions

The simulations described in this section were computed using WSNet [3]. WSNet is an event-driven simulator for wireless networks. We adapt our algorithm from the shared memory model to the message-passing model using techniques similar to those given in [13].

In this simulator, processes are randomly deployed on a square plane. Processes are motionless and equipped with radio. We use the *unit disk* metric; two processes u and v can communicate if and only if their Euclidean distance is at most *rad*, where *rad* is the transmission range. For simplicity, we consider physical and MAC layers to be ideal: there are neither interferences nor collisions. However, as stated in in [13], our algorithm still works assuming fair lossy links. Moreover, process executions are concurrent and asynchronous.

In our simulations, we consider connected UDG networks of size n between 50 and 400. They are deployed using a uniform random distribution of processes on a $100m$ side square. We tune the transmission range according to the number of processes to control the average degree $\overline{d}$ of the network. For example, by fixing n to 200 and tuning the transmission range between $14m$ and $26m$, we obtain an average degree $\overline{d}$ which varies between 10 and 50. Finally, we let k vary between 1 and 6.

The performance of $\mathcal{SMDS}(k)$ may differ depending on the spanning tree construction we used. Hence, we test our protocol using three different spanning tree constructions: depth-first spanning tree (DFS tree) [8], breadth-first spanning tree (BFS tree) [21], and arbitrary spanning tree [7].

7.2 Motivation

In the context of sensors and *ad hoc* networks, it is interesting to study average performance of the algorithms $\mathcal{DS}(k)$ and $\mathcal{MIN}(k)$ in random topologies, not just the worst case. In particular,

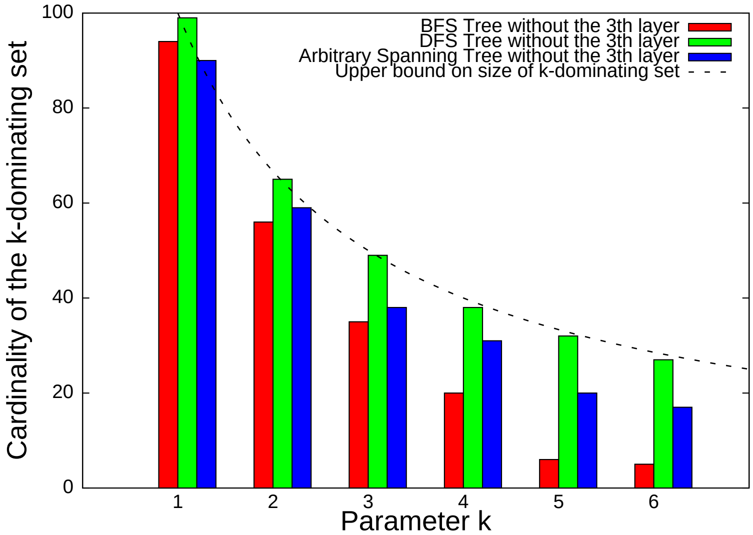

Figure 4: Average size of k-dominating set *vs.* k before minimization ($n = 200$ and $\overline{d} \in]10, 20[$)

does the choice of spanning tree make a difference in the size of k-dominating set built by $\mathcal{DS}(k)$ or $\mathcal{MIN}(k) \circ \mathcal{DS}(k)$? What is the gain due to $\mathcal{MIN}(k)$? Is this gain the same for all spanning trees? How does the size of the output k-dominating set depend on k, n, and $\overline{d}$?

7.3 Results

In this section, we summarize the performance of our algorithm in terms of the size of the k-dominating-set built by $\mathcal{DS}(k)$ and $\mathcal{MIN}(k) \circ \mathcal{DS}(k)$ in random topologies, varying k, n, $\overline{d}$ and the chosen spanning tree. For each setting (where k, n, $\overline{d}$ and the spanning tree are fixed) we made 50 runs.

Figure 4 shows the size of k-dominating set versus k after stabilization of algorithm $\mathcal{DS}(k)$. We observe that there is a noticeable difference between the computed k-dominating sets for the different kinds of spanning tree. The DFS tree, by construction, causes the number of processes in the k-dominating set to be large. We remark that the average size obtained by simulation is close to the theoretical upper bound. On the other hand, the k-dominating sets built using arbitrary and on BFS trees have better performances. The height of the tree also has a major impact on the size of the k-dominating set.

The impact of the average degree can be observed in Figure 6. The size of the k-dominating set built using a DFS tree does not change, but the size decreases if we use a BFS or an arbitrary spanning tree. When the average degree increases, the diameter of the network decreases. In the case of BFS and arbitrary spanning trees, that leads to a decrease of height, thus a decrease of the size of the k-dominating set.

Figures 4 and 6 show that the size of the k-dominating sets built by $\mathcal{DS}(k)$ in random UDGs are not far from the worst case, regardless of the tree which is used. In this context, it is interesting to study whether $\mathcal{MIN}(k)$ is able to reduce the size of the k-dominating set computed by $\mathcal{DS}(k)$ significantly.

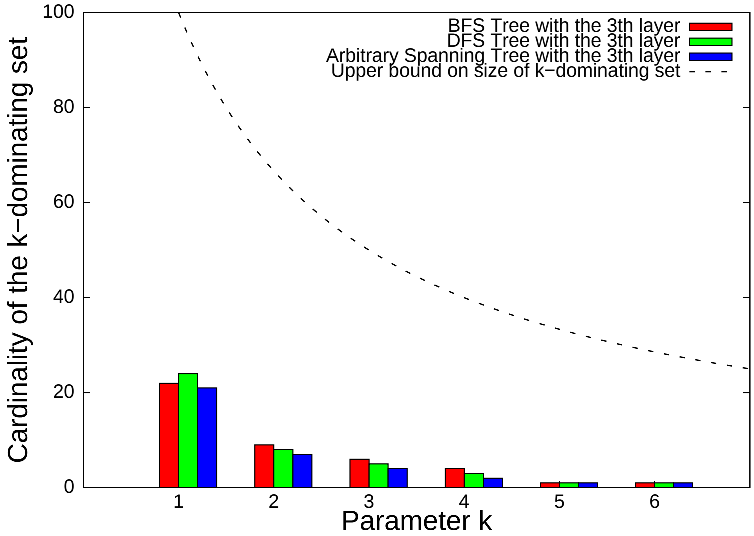

Figure 5: Average size of k-dominating set *vs.* k after minimization ($n = 200$ and $\overline{d} \in]10, 20[$)

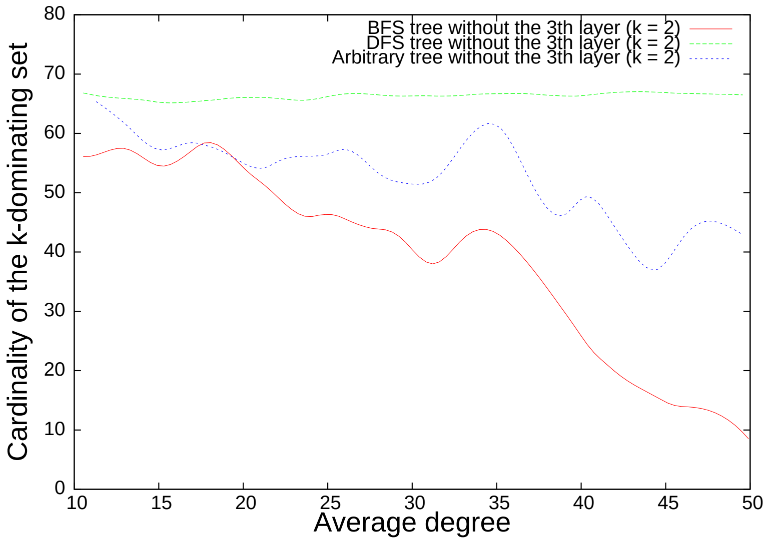

Figure 6: Average size of k-dominating set *vs.* $\overline{d}$ before minimization ($k = 2$ and $n = 200$)

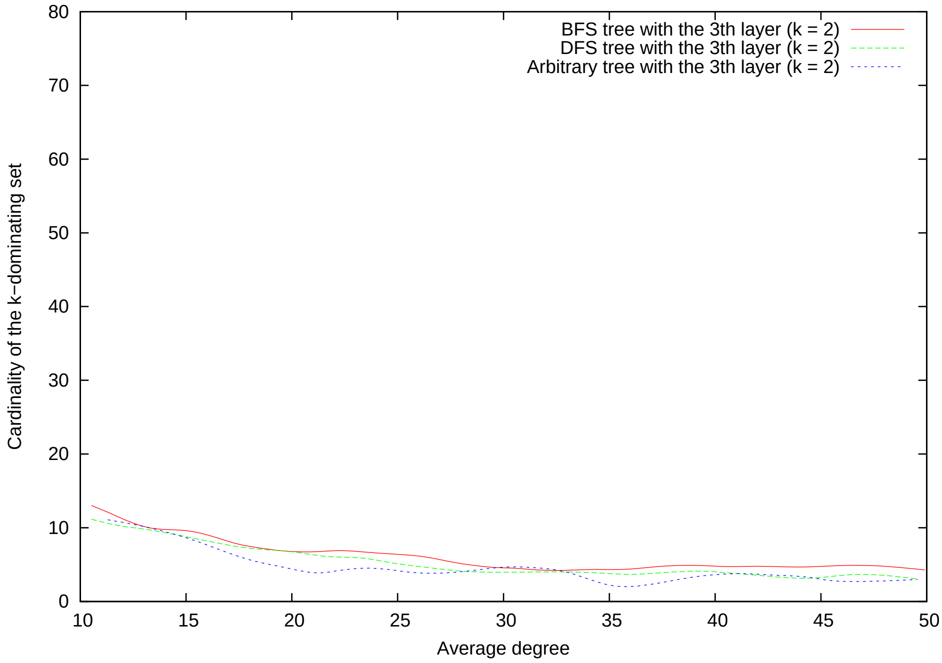

Figure 7: Average size of k-dominating set *vs.* $\overline{d}$ after minimization ($k = 2$ and $n = 200$)

Tree	Before the 3rd layer	After the 3rd layer	average gain
BFS	56.93	9.93	83%
DFS	65.87	8.93	86%
Arbitrary	59.17	7.83	87%

Table 1: Average gain of minimization

Figure 5 illustrates both the gain obtained in terms of size of k-dominating set and the differences among the k-dominating sets according to the tree for which algorithm $\mathcal{MIN}(k)$ is applied. For the three spanning tree constructions and for $1 \leq k \leq 6$, the overall average reduction is more than 75%. For higher values of k, the good performance of $\mathcal{DS}(k)$ using BFS trees prevents large gains using $\mathcal{MIN}(k)$. Here, the size of k-dominating set obtained by $\mathcal{MIN}(k)$ is quite similar for all spanning trees considered, with a slight advantage for the arbitrary spanning tree.

For $k = 2$, Figure 7 shows variations of the size of k-dominating set versus $\overline{d}$. $\mathcal{MIN}(k)$ uniformly improves the size of the k-dominating sets regardless of $\overline{d}$.

In summary, our simulations establish that the size of the computed k-dominating set is not uniformly influenced by the types of the trees used when $\mathcal{DS}(k)$ is deployed. $\mathcal{MIN}(k)$ works very well on all three kinds of trees. For example, Table 1 shows the average gain decrease of the size of the k-dominating sets computed by $\mathcal{DS}(k)$ for $k = 2$, $n = 200$, and $\overline{d}$ in $]10, 20[$.

Finally, over all simulations we made, we observe that our three-layer algorithm computes minimal k-dominating sets that are on the average drastically smaller than the theoretical bound; see, for example Figure 5. More precisely, for $n = 200$, $1 \leq k \leq 6$, and $\overline{d} \in]10, 20[$, the size of k-dominating sets we obtain, is on an average 89% smaller than the theoretical bound.

8 Conclusion

In this paper, we gave a distributed asynchronous silent self-stabilizing algorithm for finding a minimal k-dominating set of size at most $\lceil \frac{n}{k+1} \rceil$ in an arbitrary network with unique identifiers. This algorithm works under a weakly fair daemon. We then give a transformer, and use it to modify the algorithm so that it works under the unfair daemon. Using this transformer, our solution remains silent, stabilizes in $O(n)$ rounds and $O(\mathcal{D}n^3)$ steps, and uses $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, where $\mathcal{D}$ is diameter of the network and N is an upper bound on n. Our experimental results show that the size of the k-dominating set obtained by our solution is usually much smaller than $\lceil \frac{n}{k+1} \rceil$.

A possible immediate extension of our results is to determine whether it is possible to decrease the stabilization time to $O(k)$ rounds (the optimal). Another possible future research topic is to attempt to find a distributed self-stabilizing algorithm for computing a minimal k-dominating set whose size is a constant multiple of the minimum, that is, an algorithm that computes a minimal k-dominating set with a size s such that $\frac{s}{s_{opt}} \leq c$ where c is a constant and s_{opt} is the size of the minimum k-dominating set of the network.

Acknowledgments

We are grateful to the anonymous reviewers for constructive feedback and insightful suggestions which greatly improved this article.

References

[1] A D Amis, R Prakash, D Huynh, and T Vuong. Max-min d-cluster formation in wireless ad hoc networks. In *IEEE INFOCOM*, pages 32–41, 2000.

[2] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *WSS*, pages 19–34, 2001.

[3] Elyes Ben Hamida, Guillaume Chelius, and Jean-Marie Gorce. Scalable versus accurate physical layer modeling in wireless network simulations. *pads*, 0:127–134, 2008.

[4] Christian Boulinier. *L'Unisson*. PhD thesis, Université de Picardie Jules Verne, Amiens, October 2007. Dissertation in french.

[5] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *PODC*, pages 150–159, 2004.

[6] Eddy Caron, Ajoy K. Datta, Benjamin Depardon, and Lawrence L. Larmore. A self-stabilizing k-clustering algorithm for weighted graphs. *JPDC*, 70(11):1159–1173, 2010.

[7] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Inf. Process. Lett.*, 39:147–151, 1991.

[8] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Inf. Process. Lett.*, 49:297–301, 1994.

[9] Alain Cournier, Ajoy K. Datta, Franck Petit, and Vincent Villain. Enabling snap-stabilization. In *ICDCS*, pages 12–19, 2003.

[10] Ajoy K. Datta, Stéphane Devismes, and Lawrence L. Larmore. A self-stabilizing o(n)-round k-clustering algorithm. In *SRDS*, pages 147–155, 2009.

[11] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. A Self-Stabilizing O(k)-Time k-Clustering Algorithm. *The Computer Journal*, page bxn071, 2009.

[12] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theor. Comput. Sci.*, 412(40):5541–5561, 2011.

[13] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. In *Self-Stabilizing Systems*, pages 68–80, 2005.

[14] Stéphane Devismes and Franck Petit. On the complexity of the boulinier et al′s unison algorithm. Technical Report TR-2012-9, Verimag Research Report, 2012.

[15] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.

[16] Shlomi Dolev. *Self-stabilization.* MIT Press, 2000.

[17] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. In *PODC*, pages 27–34, 1996.

[18] Y Fernandess and D Malkhi. K-clustering in wireless ad hoc networks. In *ACM POMC 2002*, pages 31–37, 2002.

[19] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[20] Ted Richard Herman. *Adaptivity through distributed convergence.* PhD thesis, Austin, TX, USA, 1992. UMI Order No. GAX92-12547.

[21] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41:109–117, 1992.

[22] Michiyo Ikeda, Sayaka Kamei, and Hirotsugu Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *PDCAT*, pages 70–74, 2002.

[23] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[24] Adrian Kosowski and Lukasz Kuszner. Energy optimisation in resilient self-stabilizing processes. In *symposium on Parallel Computing in Electrical Engineering*, pages 105–110, 2006.

[25] Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *PODC*, pages 238–251, 1995.

[26] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *JACM*, 36(3):510–530, 1989.

[27] Lucia Draque Penso and Valmir C. Barbosa. A distributed algorithm to find k-dominating sets. *Discrete Appl. Math.*, 141(1-3):243–253, 2004.

[28] Vlady Ravelomanana. Distributed k-clustering algorithms for random wireless multihop networks. In *ICN (1)*, pages 109–116, 2005.

[29] S. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Observations on self-stabilizing graph algorithms on anonymous networks. In *WSS*, pages 7.1–7.15, 1995.

[30] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2nd edition, 2001.