Self-Stabilizing Small k-Dominating Sets

Stéphane Devismes, Karel Heurtefeux, Yvan Rivierre
VERIMAG Lab., Université Joseph Fourier,
Grenoble, France
e-mail: Firstname.Lastname@imag.fr

Ajoy K. Datta, Lawrence L. Larmore
School of Computer Science, University of Nevada,
Las Vegas, USA
e-mail: Firstname.Lastname@unlv.edu

***Abstract*—A self-stabilizing algorithm, after transient faults hit the system and place it in some arbitrary global state, recovers in finite time without external (*e.g.*, human) intervention. In this paper, we propose 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 propose a transformer that allows our algorithm work under an unfair daemon (the weakest scheduling assumption). The complexity of our solution is in $O(n)$ rounds and $O(\mathcal{D}n^2)$ steps using $O(\log n + k \log \frac{n}{k})$ bits per process where $\mathcal{D}$ is the diameter of the network.**

***Keywords*-Distributed systems, self-stabilization, k-dominating sets, k-clustering.**

I. Introduction

Consider a simple connected undirected graph $G = (V, E)$, where V is a set of nodes and E a set of edges. For any process 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 graph is useful because it allows to split the graph into k-clusters. A *k-cluster* of a graph is a non-empty subgraph C of radius at most k, *i.e.*, where all members of C are within distance k of the *clusterhead* of 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 given k-clustering is a k-dominating set; conversely, if D is a k-dominating set, a k-clustering is obtained by having every node choose its closest member in D as its clusterhead.

A major application of *k-clustering* is in implementing 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 [20]. We can instead consider the problem of finding a *minimal* k-dominating set, a k-dominating set D is *minimal* if for all $D' \subsetneq D$, D' is not a k-dominating set. In other words, a k-dominating set has no proper subset which is also k-dominating. However, the minimal property does not guarantee that the 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 nodes is also a minimal 1-dominating set. To overcome this problem, we propose 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.

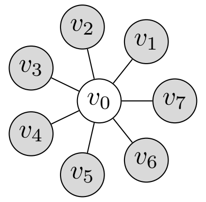


Fig. 1: Example of minimal 1-dominating set

A. Related Work

Self-stabilization [16], [17] is a versatile property, enabling an algorithm to withstand transient faults in a distributed system. A distributed algorithm is self-stabilizing if, after transient faults hit the system and place it in some arbitrary global state, the system recovers without external (*e.g.*, human) intervention in finite time.

There exist several asynchronous self-stabilizing distributed algorithms for finding a k-dominating set of a network, *e.g.*, [14], [11], [6]. All these algorithms are proven assuming an unfair daemon. The solution in [14] stabilizes in $O(k)$ rounds using $O(k \log n)$ space per process. The one in [11] stabilizes in $O(n)$ rounds using $O(\log n)$ space per process. The algorithm proposed in [6] stabilizes in $O(kn)$ rounds using $O(k \log n)$ space per process. Note that only the algorithm in [11] 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.*, [30], [23]. However, the generalization of 1-dominating set solutions to k-dominating set solutions does not scale up, 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 of a network [27], [26], [1], [19], [29]. Deterministic solutions proposed in [1], [19] 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. Solution proposed in [19] is an approximation algorithm with $O(k)$ worst case ratio over the optimal solution. The time and space complexities of the distributed algorithm in [19] are not given. In [27], 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 a complexity in $O(k \log^* n)$ time. However, the authors missed one special case in the proof, which unfortunately can make the proof fail in some networks. The same flaw is present in a few subsequent papers [26], [28]. Ravelomanana [29] proposes 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* [24], [9]. However, the transformed self-stabilizing solutions are expected to be inefficient, both in time and space, because those transformers use some mechanisms like snapshots.

B. Contributions

In this paper, we propose 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 any arbitrary identified network.

We first consider the upper bound on the size of minimum k-dominating sets given in [27]. We show that the proof given in [27] missed a case, and propose a correction that does not change the bound.

Next, we propose an asynchronous silent self-stabilizing algorithm, called $\mathcal{SMDS}(k)$, for finding a minimal k-dominating set of small size based on our proof of the bound. To simplify the design of our algorithm, we make it as a composition of four layers. The first three layers together compute a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. As the resulting k-dominating set may not be minimal, we apply the algorithm given in [11] as the fourth layer to remove nodes from D until we obtain a minimal k-dominating set. The four layer composed algorithm is proven assuming a weakly fair daemon. The solution stabilizes in $O(n)$ rounds using $O(\log n + k \log \frac{n}{k})$ bits per process, where n is the size of the network.

We then propose a general method to *efficiently* transform a self-stabilizing weakly fair algorithm into a self-stabilizing algorithm working under an unfair daemon (the weakest scheduling assumption). The proposed transformer has several advantages over the previous solutions. (1) It preserves the silence property. (2) It does not degrade the round complexity or the memory requirement of the input algorithm. (3) It builds efficient algorithms in terms of step complexity ($O(\mathcal{D}n \times R)$, 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 $O(\mathcal{D}n^2)$ steps, where $\mathcal{D}$ is the diameter of the network.

Finally, we analyse, 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 observed a noticeable gain in the size after the minimization performed by the fourth (or the last) layer.

C. Roadmap

In the next section, we present the computational model used in this paper. In Section III, we give a counterexample for the proof of the upper bound given in [27], and propose a correction. In Section IV, we present a composition technique. This technique is used to build our self-stabilizing algorithm, Algorithm $\mathcal{SMDS}(k)$, which is presented in Section V. In Section VI, we show how to transform Algorithm $\mathcal{SMDS}(k)$ to obtain a solution that works under an unfair daemon. Section VII is used to report the simulation results. We make concluding remarks in Section VIII.

Due to the lack of space, many technical proofs have been omitted. See the technical report for details [10] (http://www-verimag.imag.fr/TR/TR-2011-6.pdf).

II. Preliminaries

A. Computational Model

We consider networks as simple connected undirected graphs $G = (V, E)$, where V is a set of n processes and E a set of bidirectional links. Processes are assumed to have distinct identifiers. In the following, 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 [16]. 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*. We call *(distributed) algorithm* $\mathcal{A}$ a collection of n *programs*, each one operating on a single process. 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. 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 $\mathcal{A}$ is an instance of the states of processes in $\mathcal{A}$. We denote by $\gamma(p)$ the state of process p in configuration γ.

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 $e = \gamma_0\gamma_1 \ldots \gamma_i \ldots$ 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 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; this model is called *composite atomicity* [17].

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. A scheduler may have some *fairness* properties. Here, we consider two kinds of fairness properties. A scheduler is *weakly fair* if it allows every *continuously* enabled process to eventually execute an action. The *unfair* scheduler models designing of an algorithm with the weakest fairness assumption: it can forever prevent a process to execute an action except if the process 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 between these two configurations. The neutralization of a process represents the following situation: at least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively makes the guard of all actions of p false.

We use the notion of *round*. The first *round* of an execution ϱ, noted ϱ', is the minimal prefix of ϱ in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let ϱ'' be the suffix of ϱ starting from the last configuration of ϱ'. The second round of ϱ is the first round of ϱ'', the third round of ϱ is the second round of ϱ'', and so forth.

B. Self-Stabilization and Silence

A configuration *conforms* to a predicate if the predicate is satisfied in the configuration; otherwise the configuration *violates* the predicate. By this definition, every configuration conforms to predicate $true$, and none conforms to predicate $false$. Let R and S be predicates on configurations of the algorithm. Predicate R is *closed* with respect to the algorithm actions if every configuration of any execution of the algorithm, that starts in a configuration conforming to R, also conforms to R. Predicate R *converges* to S if R and S are closed, and every execution starting from a configuration conforming to R contains a configuration conforming to S.

A distributed algorithm is *self-stabilizing* [16] *with respect to* predicate R if $true$ converges to R. Any configuration conforming to R is said to be *legitimate*, and other configurations are called *illegitimate*.

We say that an algorithm is *silent* [18] 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.

III. 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 [27]. However, the proof proposed in [27] overlooked a special case. The same case was overlooked in some other subsequent work as well [26], [28]. Below, we exhibit a counterexample to show the special case where the proof of [27] 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. The *height* of T, noted $h(T)$, is equal to $\max_{p \in V_T} h(p)$. By extension, we denote by $h(T(p))$ the height of the subtree rooted at p, $T(p)$.

The original proof consists in dividing the processes of V into levels $T_0, \ldots, T_h$ according to their height in the tree, and assigning all the processes of height i to T_i. These sets are merged into $k+1$ sets $D_0, \ldots, D_k$ by taking $D_i = \bigcup_{j \geq 0} T_{i+j(k+1)}$.

When $k < h$, the proof in [27] 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. 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; $\|u, w\| = 3$.

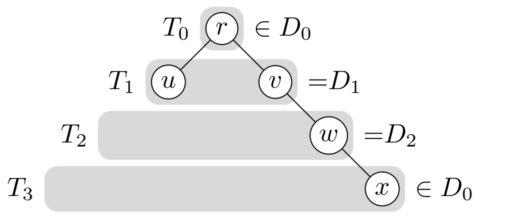


Fig. 2: Counterexample of the original proof

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_j, is not D_0. In this case, a leaf process whose height is strictly less than j may be not k-dominated by any process in D_j (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_0. Otherwise, the size of the smallest D_i ($i \in [0..k]$), say D_j, is strictly less than $\lceil \frac{n}{k+1} \rceil$ and $D_j \cup \{r\}$ is k-dominating set.

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 an arbitrary rooted spanning tree of G, and denote its height by h. Consider the $k+1$ sets $D_0, \ldots, D_k$, as previously defined.

- Assume that $k \geq h$. Then, D_0 only contains the root, and every other process is within distance k of the root. 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 $i \in [0..k]$, $|D_i| > 0$.
  - Assume that for every $i \in [0..k-1]$, $|D_i| = |D_{i+1}|$. Then, for every $i \in [0..k]$, $|D_i| = \lceil \frac{n}{k+1} \rceil$. Consider a process $v \notin D_0$. Then, the height of v, $h(v)$, satisfies $h(v) \bmod (k+1) \neq 0$. Let u be the ancestor of v such that $h(u) = h(v) - (h(v) \bmod (k+1))$ (such a process exists because $h(v) \geq (h(v) \bmod (k+1))$). Then, as $h(v) \bmod (k+1) \leq k$, u is within distance k from v.
    Remark that $h(v) = \lfloor \frac{h(v)}{k+1} \rfloor \times (k+1) + (h(v) \bmod (k+1))$. So, $h(u) = \lfloor \frac{h(v)}{k+1} \rfloor \times (k+1)$ and $h(u) \bmod (k+1) = 0$, *i.e.*, $u \in D_0$.
    Hence, D_0 is a k-dominating set such that $|D_0| = \lceil \frac{n}{k+1} \rceil$.
  - Assume that there exists $i \in [0..k-1]$, $|D_i| \neq |D_{i+1}|$. Let $j \in [0..k]$ such that $\forall i \in [0..k]$, $|D_j| \leq |D_i|$. Then, $|D_j| < \lceil \frac{n}{k+1} \rceil$. Let $D = D_j \cup \{r\}$ where r is a root of T. Then, $|D| \leq \lceil \frac{n}{k+1} \rceil$. Consider a process $v \notin D$.
    - If $h(v) \leq k$, then v is within distance k from r and $r \in D$.
    - If $h(v) > k$, then the height of v, $h(v)$, satisfies $h(v) \bmod (k+1) \neq j$. Let u be the ancestor of v such that $h(u) = h(v) - ((h(v) - j) \bmod (k+1))$. As $h(v) > k$ and $(h(v) - j) \bmod (k+1) < k+1$, $h(u) \geq 1$. Thus, u exists. Moreover, $h(v) - h(u) = (h(v) - j) \bmod (k+1) \leq k$, so v is within distance k from u. Finally, $h(u) \bmod (k+1) = (h(v) - ((h(v) - j) \bmod (k+1))) \bmod (k+1) = ((h(v) \bmod (k+1)) - ((h(v) - j) \bmod (k+1))) \bmod (k+1) = ((h(v) - (h(v) - j)) \bmod (k+1)) \bmod (k+1) = j \bmod (k+1)$. As $j \leq k$, $h(u) \bmod (k+1) = j$. So, $u \in D_j$, *i.e.*, $u \in D$.

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

□

IV. Hierarchical Collateral Composition

To simplify the design of our algorithm we use a variant of the well-known *collateral composition* [31]. Roughly speaking, when we collaterally compose two algorithms $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ and $\mathcal{B}$ run concurrently and $\mathcal{B}$ uses the outputs of $\mathcal{A}$ in its computations. In the variant we use, we modify the code of $\mathcal{B}$ so that a process executes an action of $\mathcal{B}$ only when it has no enabled action in $\mathcal{A}$.

Definition 1 (Hierarchical Collateral Composition) *Let $\mathcal{A}$ and $\mathcal{B}$ be two algorithms such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. The* hierarchical collateral composition *of $\mathcal{A}$ and $\mathcal{B}$, noted $\mathcal{B} \circ \mathcal{A}$, is the algorithm defined as follows:*

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

Below, we give a property of the *hierarchical collateral composition* (Theorem 2) that states sufficient conditions to show the correctness of the composite algorithm. For space consideration, the proofs of Theorem 2 have been omitted, see the technical report for details [10].

Theorem 2 *$\mathcal{B} \circ \mathcal{A}$ stabilizes to SP under a weakly fair daemon if the following conditions hold:*

- *$\mathcal{A}$ is silent under a weakly fair daemon.*
- *$\mathcal{B}$ stabilizes under a weakly fair daemon to SP from any configuration where no action of $\mathcal{A}$ is enabled.*[1]

V. 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 arbitrary network with nodes with unique ID's, assuming a weakly fair daemon. This algorithm is a hierarchical collateral composition of four silent self-stabilizing algorithms, $\mathcal{SMDS}(k) = \mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST} \circ \mathcal{LE}$ where:

- $\mathcal{LE}$ is a leader election algorithm.
- $\mathcal{ST}$ builds a spanning tree rooted at the process elected by $\mathcal{LE}$.
- $\mathcal{DS}(k)$ computes a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes based on the spanning tree built by $\mathcal{ST}$.
- $\mathcal{MIN}(k)$ reduces the k-dominating set built by $\mathcal{DS}(k)$ to a minimal one.

We give more details about the four layers of $\mathcal{SMDS}(k)$ in Subsections V-A to V-D. The complexity of $\mathcal{SMDS}(k)$ is presented in Subsection V-E.

A. Algorithm $\mathcal{LE}$

$\mathcal{LE}$ is any silent self-stabilizing leader election algorithm for arbitrary identified networks, assuming a weakly fair daemon. In the following, we assume the existence of the output predicate $IsLeader_p$, defined for processes p, such that $IsLeader_p$ holds if p believes to be the leader. So, $\mathcal{LE}$ converges to the predicate $SP_{\mathcal{LE}}$ defined as follows: $SP_{\mathcal{LE}}$ holds if and only if there exists a unique process p such that $IsLeader_p$. In the literature, there are several silent self-stabilizing leader election algorithms that work under a weakly fair daemon [2], [13], [12]. Here, we propose to use the algorithm given in [13]. 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 n.

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

B. Algorithm $\mathcal{ST}$

$\mathcal{ST}$ is any silent self-stabilizing spanning tree algorithm for arbitrary rooted networks, assuming a weakly fair daemon. $\mathcal{ST}$ uses the output of $\mathcal{LE}$ to decide the root of its spanning tree. In other words, $\mathcal{ST}$ builds a spanning tree rooted at the process elected by $\mathcal{LE}$. In the following, we assume that the output of $\mathcal{ST}$ is a macro called $\texttt{Parent}_p$, which is defined for all processes p. $\texttt{Parent}_p$ returns $\perp$ if p believes to be the root of the spanning tree, otherwise $\texttt{Parent}_p$ designates a neighbor q as the parent of p in the spanning tree. So, $\mathcal{ST} \circ \mathcal{LE}$ converges to the predicate $SP_{\mathcal{ST}}$ defined as follows: $SP_{\mathcal{ST}}$ holds if and only if 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.

Many silent self-stabilizing spanning tree algorithms designed for arbitrary rooted networks and working under a weakly fair daemon have been proposed in the literature. See [21] for a good survey on this topic. One of the first papers on that topic provides an algorithm to build an arbitrary spanning tree [7]. Since then, numerous algorithms have been published on various types of spanning trees, e.g., depth-first spanning tree [8], breadth-first spanning tree [22]. In our simulations, we tested our solution with each of the above three spanning tree algorithms.

From Theorem 2, we can deduce the following lemma:

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

C. Algorithm $\mathcal{DS}(k)$

$\mathcal{DS}(k)$ (see Algorithm 1 for the formal description) 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 proposed in the proof of Theorem 1 (page 4). Informally, $\mathcal{DS}(k)$ uses the following three variables at each process p:

- *p.color* $\in [0..k]$. In this variable, p computes $h(p) \bmod (k+1)$ (that is its height in T modulus $k+1$) in a top-down fashion using Action Fix-Color. Hence, once $\mathcal{DS}(k)$ has stabilized, each set D_i, defined in Section III, 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, we choose the smallest index). Then, the value of $r.min$ is broadcast in the tree.

According to Theorem 1 (page 4), 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, $\mathcal{DS}(k) \circ \mathcal{ST} \circ \mathcal{LE}$ converges to the predicate $SP_{\mathcal{DS}(k)}$ defined as follows: $SP_{\mathcal{DS}(k)}$ holds if and only if the set $\{p \in V \mid IsDominator_p = true\}$ is a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes.

Due to the lack of space, the proof of the following theorem has been omitted, see the technical report for details [10].

Theorem 3 *$\mathcal{DS}(k) \circ \mathcal{ST} \circ \mathcal{LE}$ stabilizes to $SP_{\mathcal{DS}(k)}$ in $O(n)$ rounds under a weakly fair daemon.*

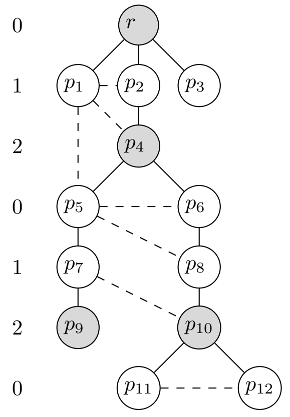


Fig. 3: Example of 2-dominating set computed by our algorithm

Figure 3 shows an example of a 2-dominating set computed by $\mathcal{DS}(2) \circ \mathcal{ST} \circ \mathcal{LE}$. In the figure, bold lines represent tree-edges, and dashed lines indicate non-tree-edges. In this example, once $\mathcal{DS}(2) \circ \mathcal{ST} \circ \mathcal{LE}$ has stabilized, $r.pop[0] = 5$, $r.pop[1] = 5$, and $r.pop[2] = 3$. Thus, $r.min = 2$, which means that the smallest used color is 2. $D_2 = \{p_4, p_9, p_{10}\}$ and $|D_2| = 3$. In this case, the 2-dominating set that $\mathcal{DS}(2) \circ \mathcal{ST} \circ \mathcal{LE}$ 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 4). 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. Also, note that this latter set is minimal because none of its proper subsets is a 2-dominating set.

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

```
Inputs:
  $Parent_p \in \mathcal{N}_p \cup \{\bot\}$      Parent process of $p$ in the spanning tree,
                                or $\bot$ for root.

Variables:
  $p.color \in [0..k]$                Color of $p$.
  $p.pop[i] \in \mathbb{N}, \forall i \in [0..k]$   Population of color $i$,
                                in the subtree rooted at $p$.
  $p.min \in [0..k]$                  Color with smallest population, in $V$.

Macros:
  $EvalColor_p$    =  $0$ if $(Parent_p = \bot)$
                      else $(Parent_p.color + 1) \bmod (k+1)$
  $SelfPop_p(i)$   =  $1$ if $(p.color = i)$ else $0$
  $Children_p$     =  $\{q \in \mathcal{N}_p \mid Parent_q = p\}$
  $EvalPop_p(i)$   =  $SelfPop_p(i) + \sum_{q \in Children_p} q.pop[i]$
  $MinPop_p$       =  $\min_{i \in [0..k]} \{p.pop[i] \mid p.pop[i] > 0\}$
  $MinColor_p$     =  $\min_{i \in [0..k]} \{i \mid p.pop[i] = MinPop_p\}$
  $EvalMin_p$      =  $MinColor_p$ if $(Parent_p = \bot)$ else $Parent_p.min$

Predicates:
  $IsRoot_p$       $\equiv$  $Parent_p = \bot$
  $ColorOK_p$      $\equiv$  $p.color = EvalColor_p$
  $PopOK_p$        $\equiv$  $\forall i \in [0..k], p.pop[i] = EvalPop_p(i)$
  $MinOK_p$        $\equiv$  $p.min = EvalMin_p$
  $IsDominator_p$  $\equiv$  $IsRoot_p \vee p.color = p.min$

Actions:
  FixColor  ::  $(\neg ColorOK_p)$            $\longrightarrow$  $p.color \leftarrow EvalColor_p$
  FixPop    ::  $(ColorOK_p \wedge$            $\longrightarrow$  $\forall i \in [0..k]$,
                  $\neg PopOK_p)$                     $p.pop[i] \leftarrow EvalPop_p(i)$
  FixMin    ::  $(ColorOK_p \wedge$
                  $PopOK_p \wedge$              $\longrightarrow$  $p.min \leftarrow EvalMin_p$
                  $\neg MinOK_p)$
```

D. Algorithm $\mathcal{MIN}(k)$

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

This last layer of our algorithm can be achieved using the silent self-stabilizing algorithm $\mathcal{MIN}(k)$ given in [11]. 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 $\{p \in V \mid p.inD = true\}$ is a minimal k-dominating set of the network.

Using the output of algorithm $\mathcal{DS}(k) \circ \mathcal{ST} \circ \mathcal{LE}$ 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 nodes in the k-dominating set computed by $\mathcal{DS}(k)$. Hence, $\mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST} \circ \mathcal{LE}$ stabilizes to the predicate $SP_{\mathcal{SMDS}(k)}$ defined as follows: $SP_{\mathcal{SMDS}(k)}$ holds if and only if the set $\{p \in V \mid p.inD = true\}$ is a minimal k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes.

As $\mathcal{SMDS}(k) = \mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST} \circ \mathcal{LE}$, from Theorem 2 and Theorem 3, we can claim the following result:

Theorem 4 (Overall Correctness) *$\mathcal{SMDS}(k)$ stabilizes to $SP_{\mathcal{SMDS}(k)}$ under a weakly fair daemon.*

E. Complexity Analysis

We first consider the round complexity of $\mathcal{SMDS}(k)$. Using the algorithm of [13], the layer $\mathcal{LE}$ stabilizes in $O(n)$ rounds. After the layer $\mathcal{LE}$ has stabilized, the layer $\mathcal{ST}$ stabilizes in $O(n)$ rounds if we use the algorithm of [22], for example. Once the spanning tree is available, $\mathcal{DS}(k)$ stabilized in $O(n)$ rounds, by Theorem 3. Finally, the k-dominating set computed by the first three layers is minimized by $\mathcal{MIN}(k)$ in $O(n)$ rounds (see [11]).

Theorem 5 *$\mathcal{SMDS}(k)$ stabilizes to $SP_{\mathcal{SMDS}(k)}$ in $O(n)$ rounds.*

We now consider the space complexity of $\mathcal{SMDS}(k)$. $\mathcal{LE}$, $\mathcal{ST}$, and $\mathcal{MIN}(k)$ can be implemented using $O(\log n)$ bits per process [13], [22], [11]. $\mathcal{DS}(k)$ 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 $\lceil \frac{n}{k+1} \rceil$. So, 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 6 *$\mathcal{SMDS}(k)$ can be implemented using $O(\log n + k \log \frac{n}{k})$ bits per process.*

VI. Transformer

In the previous section, we showed that $\mathcal{SMDS}(k)$ stabilizes to $SP_{\mathcal{SMDS}(k)}$ under a weakly fair daemon. We now propose an automatic method to transform any self-stabilizing algorithm under a weakly fair daemon into a self-stabilizing algorithm 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 [3], authors propose the *cross-over* composition. Using this composition, a weakly fair algorithm can be transformed by composing it with an algorithm that is fair[2] under an unfair daemon. 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 [25], authors propose a transformer that preserves the silence of the input algorithm. Furthermore, the step complexity analysis of the transformed algorithm is given: $O(n^4 \times R)$ 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 of the step complexity).

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

Let $\mathcal{A}$ be an algorithm that stabilizes to $SP_{\mathcal{A}}$ under a weakly fair daemon. $\mathcal{A}$ has x actions. Actions of $\mathcal{A}$ 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 $\mathcal{A}$. Actually, $\mathcal{A}^t$ is obtained by composing $\mathcal{A}$ with a self-stabilizing phase clock algorithm. This latter is treated as a black box, called $\mathcal{U}$, with the following properties:

1) Every process p has an incrementing variable $p.clock$, a member of some cycling group $\mathbb{Z}_\alpha$ where α is a positive integer.
2) The phase clock is self-stabilizing under 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) There is an action $I :: Can\_Incr_p \to Incr_p$ for each process p such that, once $\mathcal{U}$ is stabilized, I is the only action that p can execute to increment its local clock. 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].

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

$\mathcal{A}^t$ is obtained by composing $\mathcal{A}$ with $\mathcal{U}$ as follows:

- $\mathcal{A}^t$ contains all variables of $\mathcal{A}$ and $\mathcal{U}$.[3]
- $\mathcal{A}^t$ contains all actions of $\mathcal{U}$ 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 $Late_p \equiv (\exists q \in \mathcal{N}_p \mid q.clock > p.clock)$

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, every process p satisfies $Stable_p$ and, once all clocks have the same value, no further action is enabled, hence the silence is preserved.

Theorem 7 *$\mathcal{A}^t$ stabilizes to $SP_{\mathcal{A}}$ under an unfair daemon.*

Theorem 8 *If $\mathcal{A}$ is silent, then $\mathcal{A}^t$ is silent.*

Below, we present the complexity of the transformed algorithm. These results assume that $\mathcal{U}$ is the algorithm of Boulinier *et al.* in [5]. Refer to [10] for the proof.

Theorem 9 *The memory requirement of $\mathcal{A}^t$ is $O(\log n) + MEM$ bits per process, where MEM is the memory requirement of $\mathcal{A}$.*

Theorem 10 *$\mathcal{A}^t$ 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.*

Theorem 11 *$\mathcal{A}^t$ stabilizes to $SP_{\mathcal{A}}$ in $O(n^2 + \mathcal{D}nR)$ steps, where R is the stabilization time of $\mathcal{A}$ in rounds.*

As a case study, $\mathcal{SMDS}(k)^t$ stabilizes to $SP_{\mathcal{SMDS}(k)}$ in $O(n)$ rounds and $O(\mathcal{D}n^2)$ steps using $O(\log n + k \log \frac{n}{k})$ bits per process by Theorems 5-6 and 9-11. This shows that our transformer does not degrade the round complexity and memory requirement while achieving an interesting step complexity.

VII. Simulations

A. Model and assumptions

All the results provided in this section are computed using WSNet [4]. WSNet is an event-driven simulator for wireless networks. We adapt our algorithm from the state model to the message-passing model using the techniques proposed in [15].

[3]As usual, we assume that $\mathcal{A}$ does not write into the variables of $\mathcal{U}$, and conversely.

Using this simulator, we deploy processes randomly on a square plane. Processes are motionless and equipped with radio. Two processes u and v can communicate if and only if the euclidean distance between them is at most rad, where rad is the transmission range. In other words, the network topology is a Unit Disk Graph (UDG). For simplicity, we consider physical and MAC layers to be ideal: there are neither interferences nor collisions. However, as stated in in [15], 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. Tuning the transmission range between $10m$ and $50m$ makes it possible to control the average degree $\overline{d}$ of the network which varies between 10 and 50. Finally, k was varied between 1 and 6.

The performance of $\mathcal{SMDS}(k)$ may differ depending on the spanning tree construction we used in the second layer. Hence, we test our protocol using three different spanning tree constructions: depth-first spanning tree (DFS tree) [8], breadth-first spanning tree (BFS tree) [22], and arbitrary spanning tree [7].

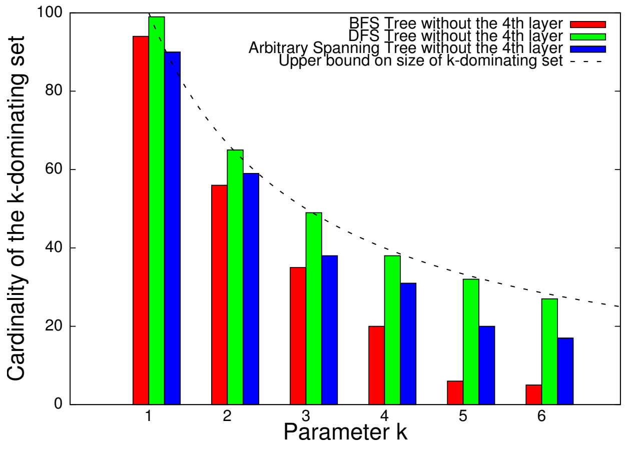
B. Motivations

In the context of sensors and ad-hoc networks, it is interesting to study average performance of algorithms $\mathcal{DS}(k)$ and $\mathcal{MIN}(k)$ in random topologies, not just the worst case. In particular, does the choice of spanning tree make a difference in terms of size of k-dominating set built by $\mathcal{DS}(k)$ or $\mathcal{DS}(k) \circ \mathcal{MIN}(k)$? 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 $\overline{d}$?

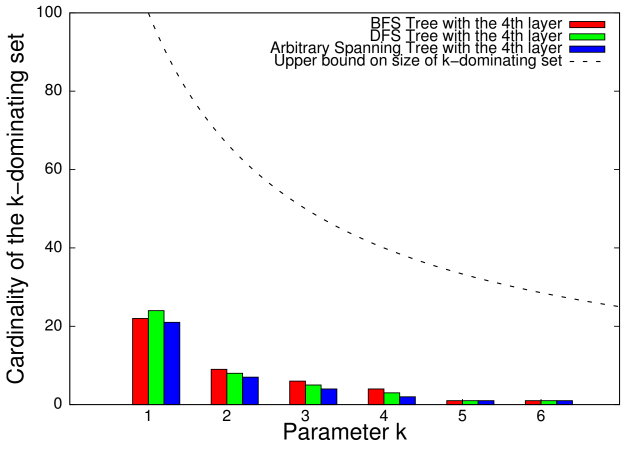
C. Results

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 $\mathcal{DS}(k) \circ \mathcal{MIN}(k)$ in random topologies, varying k, n, $\overline{d}$ and the chosen spanning tree.

Figure 4a 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 computed k-dominating sets depending on the type of the spanning tree. The DFS tree, by construction, induces a large number of k-dominating processes. We remark that the average size obtained by simulation is close to the theoretical upper bound. On the other hand, the k-dominating set built on arbitrary and BFS trees have better performances. The height of the tree also has a major impact on the size of the k-dominating set.



(a)



(b)

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

The impact of the average degree can be observed in Figure 5a. The size of the k-dominating set built on a DFS tree does not change, while it decreases the size of the ones built on 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 4a and 5a 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 they are built on. In this context, it is interesting to study if $\mathcal{MIN}(k)$ is able to reduce significantly the size of the k-dominating set computed by $\mathcal{DS}(k)$.

Figure 4b 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 on 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 $\mathcal{DS}(k)$ on BFS tree 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 5b 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 on which $\mathcal{DS}(k)$ is deployed. $\mathcal{MIN}(k)$ works very well on all the trees considered. For example, Table I shows the average gain of minimization on the k-dominating sets computed by $\mathcal{DS}(k)$ for $k = 2$, $n = 200$, and $\overline{d}$ in $]10, 20[$.

Tree	Before the 4th layer	After the 4th layer	average gain
BFS	56.93	9.93	83%
DFS	65.87	8.93	86%
Arbitrary	59.17	7.83	87%

Table I: Average gain of minimization

Finally, over all simulations we made, we observed that our four-layer algorithm computes minimal k-dominating sets that are on an average drastically smaller than the theoretical bound, see for example Figure 4b. 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.

VIII. Conclusion

In this paper, we proposed 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. We proved this algorithm assuming a weakly fair daemon. We then proposed a transformer, and used it to prove that the proposed algorithm also works under an unfair daemon. Using this transformer, our solution remains silent, stabilizes in $O(n)$ rounds and $O(\mathcal{D}n^2)$ steps, and uses $O(\log n + k \log \frac{n}{k})$ bits per process, where $\mathcal{D}$ is diameter of the network. Our experimental results show that the size of the k-dominating set obtained by our solution is usually much smaller than $\lceil \frac{n}{k+1} \rceil$.

An immediate extension of this work is to find if it is possible to enhance the stabilization time to $O(k)$ rounds (the optimal). Another future research topic is to attempt to find a distributed self-stabilizing algorithm for computing a minimal k-dominating set which is a constant approximation from the minimum one, 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.

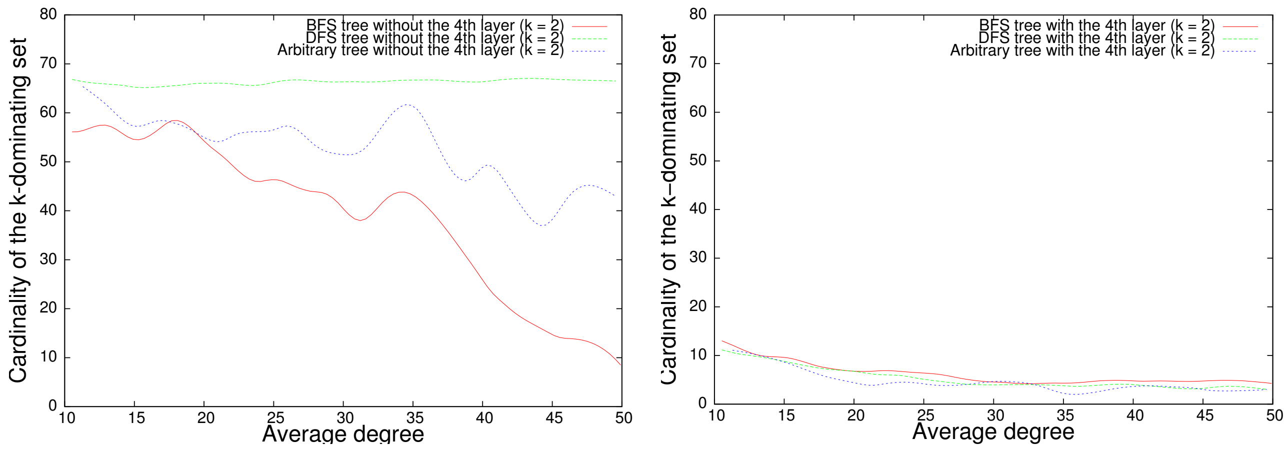


Fig. 5: Average size of k-dominating set *vs.* $\overline{d}$ ($k = 2$ and $n = 200$): (a) before; (b) after minimization

Acknowledgment

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

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] A. Arora and M. Gouda. Distributed reset. *IEEE Trans. Comput.*, 43:1026–1038, 1994.

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

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

[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, Karel Heurtefeux, Lawrence L. Larmore, and Yvan Rivierre. Self-stabilizing small k-dominating sets. Technical report, VERIMAG, 2011. http://www-verimag.imag.fr/TR/TR-2011-6.pdf.

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

[12] Ajoy K. Datta, Lawrence L. Larmore, and Hema Piniganti. Self-stabilizing leader election in dynamic networks. In *SSS*, pages 35–49, 2010.

[13] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space. In *SSS*, pages 109–123, 2008.

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

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

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

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

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

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

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

[21] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report 38, 2003.

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

[23] 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.

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

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

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

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

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

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

[30] 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.

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