



HAL
open science

Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions

Stéphane Devismes, David Ilcinkas, Colette Johnen

► **To cite this version:**

Stéphane Devismes, David Ilcinkas, Colette Johnen. Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions. 2018. hal-01667863v3

HAL Id: hal-01667863

<https://hal.science/hal-01667863v3>

Preprint submitted on 11 May 2018 (v3), last revised 9 Oct 2018 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions*

Stéphane Devismes, David Ilcinkas, and Colette Johnen

Université Grenoble Alpes, VERIMAG UMR 514, Grenoble, France
Stephane.Devismes@univ-grenoble-alpes.fr
CNRS & Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
{Ilcinkas, Johnen}@labri.fr

Abstract. We propose a general scheme, called **Algorithm Scheme**, to compute spanning-tree-like data structures on arbitrary networks. **Scheme** is self-stabilizing and silent and, despite its generality, is also efficient. It is written in the locally shared memory model with composite atomicity assuming the distributed unfair daemon, the weakest scheduling assumption of the model. Its stabilization time is in $O(n_{\max cc})$ rounds, where $n_{\max cc}$ is the maximum number of processes in a connected component. We also exhibit polynomial upper bounds on its stabilization time in steps and process moves holding for large classes of instantiations of **Algorithm Scheme**. We illustrate the versatility of our approach by proposing several such instantiations that efficiently solve classical problems.

Keywords: distributed algorithms, self-stabilization, spanning tree, leader election, spanning forest

1 Introduction

A *self-stabilizing algorithm* [1] is able to recover a correct behavior in finite time, regardless of the *arbitrary* initial configuration of the system, and therefore also after a finite number of transient faults, provided that those faults do not alter the code of the processes. Among the vast self-stabilizing literature, many works (see [2] for a survey) focus on *spanning-tree-like constructions*, *i.e.* constructions of specific distributed spanning tree- or forest- shaped data structures. Most of these constructions achieve an additional property called *silence* [3]: a silent self-stabilizing algorithm converges within finite time to a configuration from which the values of the communication registers used by the algorithm remain fixed. Silence is a desirable property. Indeed, as noted in [3], the silent property usually implies more simplicity in the algorithm design. Moreover, a silent algorithm may utilize less communication operations and communication bandwidth.

Self-stabilizing spanning-tree-like constructions are widely used as a basic building block of more complex self-stabilizing solutions. Indeed, *composition* is a natural way to design self-stabilizing algorithms [4] since it allows to simplify both the design and proofs of self-stabilizing algorithms. Various composition techniques have been introduced so far, *e.g.*, collateral composition [5], fair composition [6], cross-over composition [7], and conditional composition [8]; and many self-stabilizing algorithms are actually made as a composition of a silent spanning-tree-like construction and another algorithm designed for tree/forest topologies, *e.g.*, [9,10,11]. Notably, the silence property is not mandatory in such designs, however it allows to write simpler proofs [12]. Finally, notice that silent spanning-tree-like constructions have also been used to build very general results, *e.g.*, the self-stabilizing proof-labeling scheme constructions proposed in [13].

We consider the locally shared memory model with composite atomicity introduced by Dijkstra [1], which is the most commonly used model in self-stabilization. In this model, executions proceed in (atomic) steps and the asynchrony of the system is captured by the notion of *daemon*. The weakest (*i.e.*, the most general) daemon is the *distributed unfair daemon*. Hence, solutions stabilizing under such an assumption are highly desirable, because they work under any other daemon assumption. Moreover, the *stabilization*

*This study has been partially supported by the ANR projects DESCARTES (ANR-16-CE40-0023) and ESTATE (ANR-16-CE25-0009). This study has been carried out in the frame of “the Investments for the future” Programme IdEx Bordeaux – CPU (ANR-10-IDEX-03-02).

time can also be bounded in terms of steps (and moves, *i.e.*, local state updates) when the algorithm works under an unfair daemon. Otherwise (*e.g.*, under a weakly fair daemon), time complexity may only be evaluated in terms of rounds, which capture the execution time according to the slowest process. In contrast, step complexity captures the execution time according to the fastest process. If the average speed of the different processes are roughly equal, then the execution time is of the order of magnitude of the round complexity. Otherwise, if the system is truly asynchronous, then the execution time is of the order of magnitude of the step complexity. The stabilization time in moves captures the amount of computations an algorithm needs to recover a correct behavior. Notice that the number of moves and the number of steps are closely related: if an execution e contains x steps, then the number y of moves in e satisfies $x \leq y \leq n \cdot x$, where n is the number of processes.¹ Finally, if an algorithm is self-stabilizing under a weakly fair daemon, but not under an unfair one, then this means that the stabilization time in moves cannot be bounded, so there are processes whose moves do not make the system progress in the convergence. In other words, these processes waste computation power and so energy. Such a situation should therefore be prevented, making the unfair daemon more desirable than the weakly fair one.

There are many self-stabilizing algorithms proven under the distributed unfair daemon, *e.g.*, [14,15,16,17,18]. However, analyses of the stabilization time in steps (or moves) is rather unusual and this may be an important issue. Indeed, recently, several self-stabilizing algorithms which work under a distributed unfair daemon have been shown to have an exponential stabilization time in steps in the worst case. In [14], silent leader election algorithms from [16,17] are shown to be exponential in steps in the worst case. In [19], the Breadth-First Search (BFS) algorithm of Huang and Chen [20] is also shown to be exponential in steps. Finally, in [21] authors show that the silent self-stabilizing algorithm they proposed in [18] is also exponential in steps.

Contribution. In this paper, we propose a general scheme, called Algorithm Scheme, to compute spanning-tree-like data structures on bidirectional weighted networks of arbitrary topology (*n.b.*, the topologies are not necessarily connected). Algorithm Scheme is self-stabilizing and silent. It is written in the locally shared memory model with composite atomicity, assuming the distributed unfair daemon.

Despite its versatility, Algorithm Scheme is efficient. Indeed, its stabilization time is at most $4n_{\max\text{CC}}$ rounds, where $n_{\max\text{CC}}$ is the maximum number of processes in a connected component. Moreover, its stabilization time in moves is polynomial in the usual cases. Precisely, we exhibit polynomial upper bounds on its stabilization time in moves that depend on the particular problems we consider.

To illustrate the versatility of our approach, we propose several instantiations of Scheme solving classical spanning-tree-like problems. Assuming the network is identified (*i.e.*, processes have distinct IDs), we propose two instantiations of Scheme, for electing a leader in each connected component and building a spanning tree rooted at each leader. In one version, stabilizing in $O(n_{\max\text{CC}}^2 \cdot n)$ moves, the trees are of arbitrary topology, while trees are BFS in the other, which stabilizes in $O(n_{\max\text{CC}}^3 \cdot n)$ moves. The former move complexity matches the best known step complexity for leader election [14]. Assuming then an input set of roots, we also propose an instance to compute a spanning forest of arbitrary shaped trees, with non-rooted components detection.² This instance stabilizes in $O(n_{\max\text{CC}} \cdot n)$ moves, which matches the best known step complexity for spanning tree construction [22] with explicit parent pointers.³ Finally, assuming a rooted network, we propose a shortest-path spanning tree and DFS construction, with non-rooted components detection. The shortest-path spanning tree construction stabilizes in $O(n_{\max\text{CC}}^3 \cdot n \cdot W_{\max})$ moves (W_{\max} is the maximum weight of an edge). Again, this move complexity matches the best known move complexity for this problem [24]. From these various examples, one can easily derive other silent self-stabilizing spanning-tree-like constructions.

Related Work. This work is inspired by [24]. That paper also considers the composite atomicity model with distributed unfair daemon, is efficient both in terms of rounds and moves, tolerates disconnections,

¹Actually, in this paper as in most of the literature, bounds on step complexity are established by proving upper bounds on the number of moves.

²By non-rooted components detection, we mean that every process in a connected component that does not contain the root should eventually take a special state notifying that it detects the absence of a root.

³Actually, there exists a solution with implicit parent pointer [23] that achieves a better complexity, $O(n \cdot D)$ moves, where D is the network diameter. However adding a parent pointer to this algorithm makes this solution more costly than ours in a large class of networks, as we will explain later.

but it is restricted to the case of the shortest-path tree with a single root. Generalizing this work to obtain a generic yet efficient self-stabilizing algorithm requires a fine tuning of the algorithm (presented in Section 3) and a careful rewriting of the proofs of correctness (presented in the remaining sections). In particular, almost all the concepts used to prove termination or complexities need to be redefined to suit the new, more general setting. Consequently their new properties and the corresponding proofs are mostly novel (although of a similar flavor).

Another closely related work is the one of Cobb and Huang [25]. In that paper, a generic self-stabilizing algorithm is presented for constructing in a rooted connected network a spanning tree where a given metric is maximized. Now, since the network is assumed to be rooted (*i.e.*, a leader node is already known), leader election is not an instance of their generic algorithm. Similarly, since they assume connected networks, the non-rooted components detection cannot be expressed too. Finally, their algorithm is proven in the composite atomicity model but only for the restricted centralized weakly-fair daemon.

General schemes for arbitrary connected and identified networks have been proposed to transform almost any algorithm (specifically, those algorithms that can be self-stabilized) into their corresponding stabilizing version [26,27,28,29]. Such universal transformers are, by essence, inefficient both in terms of space and time complexities: their purpose is only to demonstrate the feasibility of the transformation. In [26] and [27], authors consider self-stabilization in asynchronous message-passing systems and in the synchronous locally shared memory model, while expressiveness of snap-stabilization is studied in [28,29] assuming the locally shared memory model with composite atomicity and a distributed unfair daemon.

In [30,31], authors propose a method to design silent self-stabilizing algorithms for a class of fix-point problems (namely fix-point problems which can be expressed using r -operators). Their solution works in directed networks using bounded memory per process. In [30], they consider the locally shared memory model with composite atomicity assuming a distributed unfair daemon, while in [31], they generalize their approach to asynchronous message-passing systems. In both papers, they establish a stabilization time in $O(D)$ rounds, where D is the network diameter, that holds for the synchronous case only.

The remainder of the related work only concerns the locally shared memory model with composite atomicity assuming a distributed unfair daemon.

In [13], authors use the concept of labeling scheme introduced by Korman *et al* [32] to design silent self-stabilizing algorithms with bounded memory per process. Using their approach, they show that every static task has a silent self-stabilizing algorithm which converges within a linear number of rounds in an arbitrary identified network. No step (nor move) complexity is given.

Efficient and general schemes for snap-stabilizing waves in arbitrary connected and rooted networks are tackled in [33]. Using this approach, one can obtain snap-stabilizing algorithms that execute each wave in a polynomial number of rounds and steps.

Few other works consider the design of particular spanning-tree-like constructions and their step complexity. Self-stabilizing algorithms that construct BFS trees in arbitrary connected and rooted networks are proposed in [34,35]. The algorithm in [34] is not silent and has a stabilization time in $O(\Delta \cdot n^3)$ steps (Δ is the maximum degree of the network). The silent algorithm given in [35] has a stabilization time $O(D^2)$ rounds and $O(n^6)$ steps. Silent self-stabilizing algorithms that construct spanning trees of arbitrary topologies in arbitrary connected and rooted networks are given in [22,23]. The solution proposed in [22] stabilizes in at most $4 \cdot n$ rounds and $5 \cdot n^2$ steps, while the algorithm given in [23] stabilizes in $n \cdot D$ moves. However, its round complexity is not analyzed and the parent of a process is not computed explicitly. Furthermore, Courrier [36] showed that the straightforward variant of this algorithm where a parent pointer variable is added has a stabilization time in $\Omega(n^2 \cdot D)$ steps in an infinite class of networks.

Several other papers propose self-stabilizing algorithms stabilizing in both a polynomial number of rounds and a polynomial number of steps, *e.g.*, [14] (for the leader election in arbitrary identified and connected networks), and [37,38] (for the DFS token circulation in arbitrary connected and rooted networks). The silent leader election algorithm proposed in [14] stabilizes in at most $3 \cdot n + D$ rounds and $O(n^3)$ steps. DFS token circulations given in [37,38] execute each wave in $O(n)$ rounds and $O(n^2)$ steps using $O(n \cdot \log n)$ space per process for the former, and $O(n^3)$ rounds and $O(n^3)$ steps using $O(\log n)$ space per process for the latter. Note that in [37], processes are additionally assumed to be identified.

Roadmap. In the next section, we present the computational model and basic definitions. In Section 3, we describe Algorithm Scheme. Its proof of correctness and a complexity analysis in moves are given in Section 4, whereas an analysis of the stabilization time in rounds is proposed in Section 5. Five

instantiations of Scheme with their specific complexity analyses are presented in Section 6. Finally, we make concluding remarks in Section 7.

2 Preliminaries

We consider *distributed systems* made of $n \geq 1$ interconnected processes. Each process can directly communicate with a subset of other processes, called its *neighbors*. Communication is assumed to be bidirectional. Hence, the topology of the system can be represented as a simple undirected graph $G = (V, E)$, where V is the set of processes and E the set of edges, representing communication links. Every (undirected) edge $\{u, v\}$ actually consists of two arcs: (u, v) (*i.e.*, the directed link from u to v) and (v, u) (*i.e.*, the directed link from v to u). For every process u , we denote by V_u the set of processes (including u) in the same connected component of G as u . In the following, V_u is simply referred to as the *connected component of u* . We denote by $n_{\max\text{CC}}$ the maximum number of processes in a connected component of G . By definition, $n_{\max\text{CC}} \leq n$.

Every process u can distinguish its neighbors using a *local labeling* of a given datatype Lbl . All labels of u 's neighbors are stored into the set $\Gamma(u)$. Moreover, we assume that each process u can identify its local label $\alpha_u(v)$ in the set $\Gamma(v)$ of each neighbor v . Such labeling is called *indirect naming* in the literature [39]. When it is clear from the context, we use, by an abuse of notation, u to designate both the process u itself, and its local labels (*i.e.*, we simply use u instead of $\alpha_u(v)$ for $v \in \Gamma(u)$). Let $\delta_u = |\Gamma(u)|$ be the *degree of process u* . The *maximal degree of G* is $\Delta = \max_{u \in V} \delta_u$.

We use the *composite atomicity model of computation* [1,6] in which the processes communicate using a finite number of locally shared registers, called *variables*. Each process can read its own variables and those of its neighbors, but can write only to its own variables. The *state* of a process is defined by the values of its local variables. A *configuration* of the system is a vector consisting of the states of each process.

A *distributed algorithm* consists of one local program per process. The *program* of each process consists of a finite set of *rules* of the form *label* : *guard* \rightarrow *action*. *Labels* are only used to identify rules in the reasoning. A *guard* is a Boolean predicate involving the state of the process and that of its neighbors. The *action* part of a rule updates the state of the process. A rule can be executed only if its guard evaluates to *true*; in this case, the rule is said to be *enabled*. A process is said to be enabled if at least one of its rules is enabled. We denote by $Enabled(\gamma)$ the subset of processes that are enabled in configuration γ .

When the configuration is γ and $Enabled(\gamma) \neq \emptyset$, a non-empty set $\mathcal{X} \subseteq Enabled(\gamma)$ is selected by the so-called daemon; then every process of \mathcal{X} *atomically* executes one of its enabled rules, leading to a new configuration γ' . The transition from γ to γ' is called a *step*. We also say that each process of \mathcal{X} performs a *move* or an *action* during γ to γ' . The possible steps induce a binary relation over \mathcal{C} , denoted by \mapsto . An *execution* is a maximal sequence of 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 in which no rule is enabled at any process.

Each step from a configuration to another is driven by a daemon. We define a daemon as a predicate over executions. We say that an execution e is an *execution under the daemon S* , if $S(e)$ holds. In this paper we assume that the daemon is *distributed* and *unfair*. “Distributed” means that while the configuration is not terminal, the daemon should select at least one enabled process, maybe more. “Unfair” means that there is no fairness constraint, *i.e.*, the daemon might never select an enabled process unless it is the only enabled process. In other words, the distributed unfair daemon corresponds to the predicate *true*, *i.e.*, this is the most general daemon.

In the composite atomicity model, an algorithm is *silent* if all its possible executions are finite. Hence, we can define silent self-stabilization as follows.

Definition 1 (Silent Self-Stabilization). *Let \mathcal{L} be a non-empty subset of configurations, called the set of legitimate configurations. A distributed system is silent and self-stabilizing under the daemon S for \mathcal{L} if and only if the following two conditions hold:*

- all executions under S are finite, and
- all terminal configurations belong to \mathcal{L} .

We use the notion of *round* [40] to measure the time complexity. The definition of round uses the concept of *neutralization*: a process v is *neutralized* during a step $\gamma_i \mapsto \gamma_{i+1}$, if v is enabled in γ_i but not in configuration γ_{i+1} , and it is not activated in the step $\gamma_i \mapsto \gamma_{i+1}$. Then, the rounds are inductively defined as follows. The first round of an execution $e = \gamma_0, \gamma_1, \dots$ is the minimal prefix $e' = \gamma_0, \dots, \gamma_j$, such that every process that is enabled in γ_0 either executes a rule or is neutralized during a step of e' . Let e'' be the suffix $\gamma_j, \gamma_{j+1}, \dots$ of e . The second round of e is the first round of e'' , and so on.

The *stabilization time* of a silent self-stabilizing algorithm is the maximum time, in moves, steps or rounds, over every execution possible under the considered daemon S (starting from any initial configuration) to reach a terminal (legitimate) configuration.

3 Algorithm Scheme

3.1 The problem

We propose a general silent self-stabilizing algorithm, called Scheme (see Algorithm 1 for its formal code), which aims at converging to a terminal configuration where a specified spanning forest (maybe a single spanning tree) is (distributedly) defined. To that goal, each process u has two inputs.

$canBeRoot_u$: a constant boolean value, which is true if u is allowed to be root of a tree. In this case, u is called a *candidate*. In a terminal configuration, every tree root satisfies $canBeRoot$, but the converse is not necessarily true. Moreover, for every connected component GC , if there is at least one candidate $u \in GC$, then at least one process of GC should be a tree root in a terminal configuration. In contrast, if there is no candidate in a connected component, we require that all processes of the component converge to a particular terminal state, expressing the local detection of the absence of candidate.

$pname_u$: the name of u (a constant). $pname_u \in IDs$, where $IDs = \mathbb{N} \cup \{\perp\}$ is totally ordered by $<$ and $\min_<(IDs) = \perp$. The value of $pname_u$ is problem dependent. Actually, we consider here two particular cases of naming. In one case, $\forall v \in V, pname_v = \perp$. In the other case, $\forall u, v \in V, pname_u \neq \perp \wedge (u \neq v \Rightarrow pname_u \neq pname_v)$, *i.e.*, $pname_u$ is a unique global *identifier*.

Then, according to the specific problem we consider, we may want to minimize the weight of the trees using some kind of distance. To that goal, we assume that each edge $\{u, v\}$ has two *weights*: $\omega_u(v)$ denotes the weight of the arc (u, v) and $\omega_v(u)$ denotes the weight of the arc (v, u) . Both values belong to the domain $DistSet$. Let $(DistSet, \oplus, \prec)$ be an ordered magma, *i.e.*, \oplus is a closed binary operation on $DistSet$ and \prec is a total order on this set. The definition of $(DistSet, \oplus, \prec)$ is problem dependent and, if necessary (*i.e.*, if the problem dependent predicate $P_nodeImp(\cdot)$ holds), the weight of the trees will be minimized using the ordered magma and the distance values that each candidate u may take when it is the root of a tree. This latter value is given by the (problem dependent) function $distRoot(u)$.

We assume that, for every edge $\{u, v\}$ of E and for every value d of $DistSet$, we have $d \prec d \oplus \omega_u(v)$ and $d \prec d \oplus \omega_v(u)$. Besides, for every $d1$ and $d2$ in $DistSet$, and for every integer $i \geq 0$, we define $d1 \oplus (i \cdot d2)$ as follows:

- $d1 \oplus (0 \cdot d2) = d1$
- $d1 \oplus (i \cdot d2) = (d1 \oplus ((i - 1) \cdot d2)) \oplus d2$ if $i > 0$.

3.2 The variables

In Scheme, each process u maintains the following three variables.

$st_u \in \{I, C, EB, EF\}$: this variable gives the *status* of the process. I , C , EB , and EF respectively stand for *Isolated*, *Correct*, *Error Broadcast*, and *Error Feedback*. The two first status, I and C , are involved in the normal behavior of the algorithm, while the two last ones, EB and EF , are used during the correction mechanism. The meaning of EB and EF will be further detailed in Subsection 3.4. In a terminal configuration, if V_u contains a candidate, then $st_u = C$, otherwise $st_u = I$.

$parent_u \in \{\perp\} \cup Lbl$: In a terminal configuration, if V_u contains a candidate, then either $parent_u = \perp$, *i.e.*, u is a tree root, or $parent_u$ belongs to $\Gamma(u)$, *i.e.*, $parent_u$ designates a neighbor of u , referred to as its *parent*. Otherwise (V_u does not contain a candidate), the value of $parent_u$ is meaningless.

$d_u \in DistSet$. In a terminal configuration, if V_u contains a candidate, then d_u is larger than or equal to the weight of the tree path from u to its tree root, otherwise the value of d_u is meaningless.

3.3 Typical Execution

Consider a configuration where, for every process u , $st_u = I$. All processes that belong to a connected component containing no candidates are disabled forever. Focus now on a connected component GC where at least one process is candidate. Then, any process u of status I that is a candidate or a neighbor of a process of status C is enabled to execute rule \mathbf{R}_R : it eventually executes $\mathbf{R}_R(u)$ to initiate a tree or to join a tree rooted at some candidate, choosing among the different possibilities the one that minimizes its distance value. Using this rule, it also switches its status to C and sets d_u to $distRoot(u)$, or $d_v \oplus \omega_u(v)$ if it chooses a parent v . Executions of rule \mathbf{R}_R are asynchronously propagated in GC until all processes of GC have status C . In parallel, rules \mathbf{R}_U are executed to reduce the weight of the trees, if necessary: when a process u with status C satisfies $P_nodeImp(u)$, this means that u can reduce d_u by selecting another neighbor with status C as parent and this reduction is required by the specification of the problem to be solved ($P_nodeImp(u)$ is problem dependent). In this case, u chooses the neighbor which allows to minimize the value of d_u . In particular, notice that a candidate can lose its tree root condition using this rule, if it finds a sufficiently good parent in its neighborhood. Hence, eventually the system reaches a terminal configuration, where a specific spanning forest (maybe a single spanning tree) is defined (in a distributed manner) in connected components containing at least one candidate, while in other components all processes are isolated.

3.4 Error Correction

Assume now that the system is in an arbitrary configuration. Inconsistencies between the states of the processes are detected using predicate $P_abnormalRoot$. We call *abnormal root* any process u satisfying $P_abnormalRoot(u)$. Informally (see Subsection 4.1, page 7, for the formal definition), a process u is an *abnormal root* if u is neither a normal root (i.e., $\neg P_root(u)$, see Definition 2), nor isolated (i.e. $st_u \neq I$), and satisfies one of the following four conditions:

1. its parent pointer does not designate a neighbor,
2. its parent has status I ,
3. its distance d_u is inconsistent with the distance of its parent, or
4. its status is inconsistent with the status of its parent.

Every process u that is neither an abnormal root nor isolated satisfies one of the two following cases. Either u is a normal root, i.e., $P_root(u)$, or u points to some neighbor (i.e., $parent_u \in \Gamma(u)$) and the state of u is coherent w.r.t. the state of its parent. In this latter case, $u \in Children(parent_u)$, i.e., u is a “real” child of its parent (see Subsection 4.1 for the formal definition). Consider a path $\mathcal{P} = u_0, \dots, u_k$ such that $\forall i, 0 \leq i < k, u_{i+1} \in Children(u_i)$. \mathcal{P} is acyclic. If u_0 is either a normal or an abnormal root, then \mathcal{P} is called a *branch* rooted at u_0 . Let u be a root (either normal or abnormal). We define the tree $T(u)$ as the set of all processes that belong to a branch rooted at u . If u is a normal root, then $T(u)$ is said to be a *normal tree*, otherwise u is an abnormal root and $T(u)$ is said to be an *abnormal tree*.

We call *normal configuration* any configuration without abnormal trees. So, to recover a normal configuration, it is necessary to remove all abnormal trees. For each abnormal tree T , we have two cases. If the abnormal root u of T can join another tree T' using rule $\mathbf{R}_U(u)$ (thus without increasing its distance value), then it does so and T disappears by becoming a subtree of T' . Otherwise, T is entirely removed in a top-down manner, starting from its abnormal root u . Now, in that case, we have to prevent the following situation: u leaves T ; this removal creates some abnormal trees, each of those being rooted at a previous child of u ; and later u joins one of those (created) trees. (This issue is sometimes referred to as the count-to-infinity problem [41].) Hence, the idea is to freeze T , before removing it. By freezing we mean assigning each member of the tree to a particular state, here EF , so that (1) no member v of the tree is allowed to execute $\mathbf{R}_U(v)$, and (2) no process w can join the tree by executing $\mathbf{R}_R(w)$ or $\mathbf{R}_U(w)$. Once frozen, the tree can be safely deleted from its root to its leaves.

The freezing mechanism (inspired from [42]) is achieved using the status EB and EF , and the rules \mathbf{R}_{EB} and \mathbf{R}_{EF} . If a process is not involved into any freezing operation, then its status is I or C . Otherwise, it has status EB or EF and no neighbor can select it as its parent. These two latter status are actually used to perform a “Propagation of Information with Feedback” [43,44] in the abnormal trees. This is why status EB means “Error Broadcast” and EF means “Error Feedback”. From an abnormal root, the

status EB is broadcast down in the tree using rule \mathbf{R}_{EB} . Then, once the EB wave reaches a leaf, the leaf initiates a convergecast EF -wave using rule \mathbf{R}_{EF} . Once the EF -wave reaches the abnormal root, the tree is said to be *dead*, meaning that all processes in the tree have status EF and, consequently, no other process can join it. So, the tree can be safely deleted from its abnormal root toward its leaves. There are two possibilities for the deletion depending on whether or not the process u to be deleted is a candidate or has a neighbor with status C . If u is a candidate or has a neighbor with status C , the rule $\mathbf{R}_R(u)$ is executed: u tries to directly join another “alive” tree, however if becoming a normal root allows it to further minimize d_u , it executes $beRoot(u)$ to become a normal root. If u is not a candidate and has no neighbor with status C , the rule $\mathbf{R}_I(u)$ is executed: u becomes isolated, and might join another tree later.

Let u be a process belonging to an abnormal tree of which it is not the root. Let v be its parent. From the previous explanation, it follows that during the correction, $(st_v, st_u) \in \{(C, C), (EB, C), (EB, EB), (EB, EF), (EF, EF)\}$ until v resets by $\mathbf{R}_R(v)$ or $\mathbf{R}_I(v)$. Now, due to the arbitrary initialization, the status of u and v may not be coherent, in this case u is an abnormal root. Precisely, as formally defined in Algorithm 1, the status of u is incoherent *w.r.t* the status of its parent v if $st_u \neq st_v$ and $st_v \neq EB$.

Actually, the freezing mechanism ensures that if a process is the root of an alive abnormal tree, it is in that situation since the initial configuration (see Lemma 5, page 10). The bounded move complexity mainly relies on this strong property.

4 Correctness and Move Complexity of Scheme

4.1 Definitions

Before proceeding with the proof of correctness and the move complexity analysis, we define some useful concepts and give some of their properties.

Root, Child, and Branch.

Definition 2 (Normal and Abnormal Roots). *Every process u that satisfies $P_root(u)$ is said to be a normal root.*

Every process u that satisfies $P_abnormalRoot(u)$ is said to be an abnormal root.

Definition 3 (Alive Abnormal Root). *A process u is said to be an alive abnormal root (resp. a dead abnormal root) if u is an abnormal root and has a status different from EF (resp. has status EF).*

Definition 4 (Children). *For every process v , $Children(v) = \{u \in \Gamma(v) \mid st_v \neq I \wedge st_u \neq I \wedge parent_u = v \wedge d_u \succeq d_v \oplus \omega_u(v) \wedge (st_u = st_v \vee st_v = EB)\}$.*

For every process $u \in Children(v)$, u is said to be a child of v . Conversely, v is said to be the parent of u .

Observation 1 *A process u is either a normal root, an isolated process (i.e. $st_u = I$), an abnormal root, or a child of its parent v (i.e. member of the set $Children(parent_v)$).*

Definition 5 (Branch). *A branch is a sequence of processes v_0, \dots, v_k , for some integer $k \geq 0$, such that v_0 is a normal or an abnormal root and, for every $0 \leq i < k$, we have $v_{i+1} \in Children(v_i)$. The process v_i is said to be at depth i and v_i, \dots, v_k is called a sub-branch. If v_0 is an abnormal root, the branch is said to be illegal, otherwise, the branch is said to be legal.*

Observation 2 *A branch depth is at most $n_{maxCC} - 1$. A process v having status I does not belong to any branch. If a process v has status C (resp. EF), then all processes of a sub-branch starting at v have status C (resp. EF).*

One of the key properties allowing us to prove that Scheme has a polynomial move complexity is the following result.

Algorithm 1: Algorithm Scheme, code for any process u

Inputs

- $canBeRoot_u$: a boolean value; it is true if u can be a root
- $pname_u$: name of u

Variables

- $st_u \in \{I, C, EB, EF\}$: the status of u
- $parent_u \in \{\perp\} \cup Lbl$
- d_u : the distance value associated to u

Predicates

- $P_root(u) \equiv canBeRoot_u \wedge st_u = C \wedge parent_u = \perp \wedge d_u = distRoot(u)$
- $P_abnormalRoot(u) \equiv \neg P_root(u) \wedge st_u \neq I \wedge [parent_u \notin \Gamma(u) \vee st_{parent_u} = I \vee d_u < d_{parent_u} \oplus \omega_u(parent_u) \vee (st_u \neq st_{parent_u} \wedge st_{parent_u} \neq EB)]$
- $P_reset(u) \equiv st_u = EF \wedge P_abnormalRoot(u)$
- $P_updateNode(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C \wedge d_v \oplus \omega_u(v) < d_u)$
- $P_updateRoot(u) \equiv canBeRoot_u \wedge distRoot(u) < d_u$
- $P_nodeImp(u)$ is problem dependent. However,
 - if $P_nodeImp(u)$, then $P_updateNode(u) \vee P_updateRoot(u)$;
 - if $P_updateRoot(u)$, then $P_nodeImp(u)$;
 - $P_nodeImp(u)$ only depends on the values of st_u , d_u , $P_updateRoot(u)$, and $\min_{(v \in \Gamma(u) \wedge st_v = C)}(d_v \oplus \omega_u(v))$.

Macros

- $beRoot(u)$: $st_u := C$; $parent_u := \perp$; $d_u := distRoot(u)$;
- $computePath(u)$:
 - $st_u := C$;
 - $parent_u := \operatorname{argmin}_{(v \in \Gamma(u) \wedge st_v = C)}(d_v \oplus \omega_u(v))$;
 - $d_u := d_{parent_u} \oplus \omega_u(parent_u)$;
 - if $P_updateRoot(u)$ **then** $beRoot(u)$;

Rules

- $\mathbf{R}_U(u)$: $st_u = C \wedge P_nodeImp(u) \rightarrow computePath(u)$;
 - $\mathbf{R}_{EB}(u)$: $st_u = C \wedge \neg P_nodeImp(u) \wedge (P_abnormalRoot(u) \vee st_{parent_u} = EB) \rightarrow st_u := EB$;
 - $\mathbf{R}_{EF}(u)$: $st_u = EB \wedge (\forall v \in Children(u) \mid st_v = EF) \rightarrow st_u := EF$;
 - $\mathbf{R}_I(u)$: $P_reset(u) \wedge \neg canBeRoot_u \wedge (\forall v \in \Gamma(u) \mid st_v \neq C) \rightarrow st_u := I$;
 - $\mathbf{R}_R(u)$: $(P_reset(u) \vee st_u = I) \wedge [canBeRoot_u \vee (\exists v \in \Gamma(u) \mid st_v = C)] \rightarrow computePath(u)$;
-

Legitimate Configurations.

Definition 6 (Legitimate State). A process u is said to be in a legitimate state of Scheme if u satisfies one of the following conditions:

1. $P_root(u)$, and $\neg P_nodeImp(u)$,
2. there is a process satisfying $canBeRoot$ in V_u , $st_u = C$, $parent_u \in \Gamma(u)$, $d_u \succeq d_{parent_u} \oplus \omega_u(parent_u)$, and $\neg P_nodeImp(u)$, or
3. there is no process satisfying $canBeRoot$ in V_u and $st_u = I$.

Definition 7 (Legitimate Configuration). A legitimate configuration of Scheme is a configuration where every process is in a legitimate state.

We will eventually prove that the terminal configurations are exactly the legitimate configurations. We first prove one of the two inclusions.

Lemma 1. Any legitimate configuration of Scheme is terminal.

Proof. Let γ be a legitimate configuration of Scheme and u be a process.

Assume first that there is no process of V_u that satisfies $canBeRoot$ in γ . Then, by definition of γ , every process v in V_u satisfies $st_v = I$. Hence, since $\neg canBeRoot_v \wedge st_v = I$ for every process v in V_u , no rule of Scheme is enabled at any process of V_u in γ .

Assume then that there is a process that satisfies $canBeRoot$ in γ . Then, every process $v \in V_u$ satisfies (1) $P_root(v)$ and $\neg P_nodeImp(v)$, or (2) $st_v = C$, $parent_v \in \Gamma(v)$, $d_v \succeq d_{parent_v} \oplus \omega_v(parent_v)$, and $\neg P_nodeImp(v)$. This in particular means that $st_v = C$, for every $v \in V_u$. Hence, $\mathbf{R}_{EF}(v)$, $\mathbf{R}_I(v)$, and $\mathbf{R}_R(v)$ are all disabled at every $v \in V_u$ in γ . $\neg P_nodeImp(v)$ implies that $\mathbf{R}_U(v)$ is disabled at every $v \in V_u$. Finally, $st_v = C \wedge [P_root(v) \vee (parent_v \in \Gamma(v) \wedge d_v \succeq d_{parent_v} \oplus \omega_v(parent_v))]$ for every $v \in V_u$ implies $\neg P_abnormalRoot(v) \wedge st_{parent_v} \neq EB$ for every $v \in V_u$ and so $\mathbf{R}_{EB}(v)$ is disabled at every $v \in V_u$ in γ . Hence, no rule of Scheme is enabled at any process of V_u in γ . \square

4.2 Partial Correctness

The following technical lemmas will help us to prove that any terminal configuration of Scheme is legitimate.

Lemma 2. In any terminal configuration of Scheme, every process has status I or C .

Proof. Assume that there exists some process that has status EB . Consider a process u with status EB having the largest distance value d_u . Note that no process v that has status C can be a child of u , otherwise $\mathbf{R}_U(v)$ or $\mathbf{R}_{EB}(v)$ would be enabled. Therefore, process u has only children having the status EF . Thus $\mathbf{R}_{EF}(u)$ is enabled, a contradiction.

Assume now that there exists some process that has status EF . Consider a process u with status EF having the smallest distance value d_u . As no process has status EB (see the previous case), u is an abnormal root, and has the status EF . So, either $\mathbf{R}_I(u)$ or $\mathbf{R}_R(u)$ is enabled, a contradiction. \square

Lemma 3. Let γ be a terminal configuration of Scheme. Let u be a process such that V_u contains at least one process satisfying $canBeRoot$ in γ . In γ , u satisfies:

- $st_u = C$,
- $\neg P_nodeImp(u)$, and
- $P_root(u)$ or $parent_u \in \Gamma(u) \wedge d_u \succeq d_{parent_u} \oplus \omega_u(parent_u)$.

Proof. Let v be a process of V_u such that $canBeRoot_v$ in γ . We have $st_v \notin \{EB, EF\}$, by Lemma 2, and also $st_v \neq I$, because otherwise $\mathbf{R}_R(v)$ would be enabled in γ . Therefore $st_v = C$.

Assume then that there exists some process of V_u that has status I in γ . Consider now a process w of V_u such that w has status I and at least one of its neighbors has status C in γ (such a process exists because no process has status EB or EF in γ , by Lemma 2, whereas at least one process, e.g., v , of V_u

has status C). Then, $\mathbf{R}_R(w)$ is enabled in γ , a contradiction. So, every process of V_u (including u) has status C in γ .

Since $st_u = C$ in γ , $\neg P_nodeImp(u)$ holds in γ (otherwise, $\mathbf{R}_U(u)$ would be enabled).

In γ , u satisfies $\neg P_abnormalRoot(u)$ because, otherwise, either $\mathbf{R}_U(u)$ or $\mathbf{R}_{EB}(u)$ would be enabled, as $st_u = C$ in γ . We can thus conclude by Observation 1 that u satisfies $P_root(u)$ or $parent_u \in \Gamma(u) \wedge d_u \succeq d_{parent_u} \oplus w_u(parent_u)$ in γ . \square

In a connected component containing no candidates, all processes are isolated. Otherwise, the process u having the smallest distance d_u (which would exist by Lemma 2) would be an abnormal root and thus would be enabled. Therefore, by Lemma 3, we obtain the following result.

Theorem 1. *Any terminal configuration of Scheme is legitimate.*

In the remainder of Section 4, we establish some properties on every execution of Scheme under a distributed unfair daemon. These properties allow us to show the termination under a distributed unfair daemon and exhibit an upper bound on the move complexity of any instance of Scheme.

4.3 GC-segments

Let GC be a connected component of G and let γ be a configuration. Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in γ , u is an alive abnormal root, or $P_updateRoot(u) \wedge st_u = C$ holds. We now prove that this set can never gain a new element.

Lemma 4. *Let $\gamma \mapsto \gamma'$ be a step where a process u executes the rule \mathbf{R}_U or \mathbf{R}_R . Then u is not an alive abnormal root in γ' .*

Proof. If $parent_u = \perp$ in γ' , then u must have executed $beRoot(u)$ in $\gamma \mapsto \gamma'$. So $P_root(u)$ is true in γ' , which, in turn, implies $\neg P_abnormalRoot(u)$.

Assume now that $parent_u = v$ in γ' . Then $st_v = C$ in γ (because it is a requirement to execute \mathbf{R}_U or \mathbf{R}_R when $beRoot(u)$ is not executed in $\gamma \mapsto \gamma'$). Consequently, the only rules that v may execute in $\gamma \mapsto \gamma'$ are \mathbf{R}_U or \mathbf{R}_{EB} . During $\gamma \mapsto \gamma'$, v either takes the status EB , decreases its distance value, or does not change the value of its variables. In any case, u belongs to $Children(v)$ in γ' , which prevents u from being an alive abnormal root in γ' . \square

Lemma 5. *No alive abnormal root is created along any execution of Scheme.*

Proof. Let $\gamma \mapsto \gamma'$ be a step. Let u be a process that is *not* an alive abnormal root in γ . If the status of u is EF or I in γ' , then u is not an alive abnormal root in γ' . If u executes \mathbf{R}_U or \mathbf{R}_R during this step, then u is not an alive abnormal root in γ' either, by Lemma 4. So the only rule that u may execute is \mathbf{R}_{EB} in $\gamma \mapsto \gamma'$. Furthermore, both in γ and γ' , u has status C or EB .

Assume first that $parent_u = \perp$ in γ' . Then, $parent_u = \perp$ already holds in γ . We thus have $P_root(u)$ in γ because $\neg P_abnormalRoot(u)$ in γ . Consequently, u executes no move in $\gamma \mapsto \gamma'$, and u is still a normal root in γ' .

Assume now that $parent_u = v$ in γ' . Whether u executes \mathbf{R}_{EB} or not, $parent_u$ is also v in γ . Also, $\neg P_abnormalRoot(u)$ in γ implies that $u \in Children(v)$ and $st_v \in \{C, EB\}$ in γ , further implying that the only rules that v may execute in $\gamma \mapsto \gamma'$ are \mathbf{R}_U or \mathbf{R}_{EB} . We conclude that, in any case, u still belongs to $Children(v)$ in γ' , which prevents u from being an alive abnormal root in γ' . \square

Lemma 6. *If a process u satisfies $P_updateRoot(u) \wedge st_u = C$, then it does so from the beginning of the execution.*

Proof. Let u be a process satisfying $P_updateRoot(u) \wedge st_u = C$. Note that the property does only depend on the local state on u . Moreover, this state must be the initial state, because any rule fixing st_u to C also sets d_u to a value not larger than $distRoot_u$. This concludes the proof. \square

By the two preceding Lemmas, we obtain the following result.

Corollary 1. *For every step $\gamma \mapsto \gamma'$, $SL(\gamma', GC) \subseteq SL(\gamma, GC)$.*

Based on Corollary 1, we can use the notion of *GC-segment* defined below to bound the total number of moves in an execution.

Definition 8 (GC-Segment). *Let $e = \gamma_0, \gamma_1, \dots$ be an execution of Scheme. Let GC be a connected component of G . If there is no step $\gamma_i \mapsto \gamma_{i+1}$ in e such that $|SL(\gamma_i, GC)| > |SL(\gamma_{i+1}, GC)|$, then the first GC-segment of e is e itself and there is no other GC-segment.*

Otherwise, let $\gamma_i \mapsto \gamma_{i+1}$ be the first step of e such that $|SL(\gamma_i, GC)| > |SL(\gamma_{i+1}, GC)|$. The first GC-segment of e is the prefix $\gamma_0, \dots, \gamma_{i+1}$. The second GC-segment of e is the first GC-segment of the suffix $\gamma_{i+1}, \gamma_{i+2}, \dots$, and so forth.

By Corollary 1, we have

Observation 3 *Let GC be a connected component of G . For every execution e of Scheme, e contains at most $n_{\max CC} + 1$ GC-segments, because $|SL(\gamma_i, GC)| \leq n_{\max CC}$ by definition.*

Let us now prove some properties on the moves made by a process in a GC-segment.

Lemma 7. *Let GC be a connected component of G and u be any process of GC . Let seg be a GC-segment. During seg , if u executes the rule \mathbf{R}_{EF} , then u does not execute any other rule in the remaining of seg .*

Proof. Let $\gamma_1 \mapsto \gamma_2$ be a step of seg in which u executes \mathbf{R}_{EF} . Let $\gamma_3 \mapsto \gamma_4$ be the next step in which u executes a rule. (If one of these two steps does not exist, then the lemma trivially holds.)

Let v be the root (at depth 0) of any branch in γ_1 containing u . By Definition 4, v must have status EB , and must therefore be an alive abnormal root. This implies that $v \in SL(\gamma_1, GC)$. Note that we may have $v = u$. On the other hand, in γ_3 , u is the dead abnormal root of all branches it belongs to since $st_u = EF$ in γ_3 and u necessarily executes \mathbf{R}_I or \mathbf{R}_R in this step. This implies that v must have executed the rule \mathbf{R}_{EF} in the meantime: there is a step $\gamma_5 \mapsto \gamma_6$, with γ_5 between γ_1 (included) and γ_3 (excluded) where v executes \mathbf{R}_{EF} . Since $st_v = EF$ in γ_6 , we have $v \notin SL(\gamma_6, GC)$. Therefore, the steps $\gamma_1 \mapsto \gamma_2$, and $\gamma_3 \mapsto \gamma_4$ belong to two distinct GC-segments of the execution, by Corollary 1 and Definition 8. \square

By Lemma 7, we obtain the following result.

Corollary 2. *Let GC be a connected component of G and u be any process of GC . The sequence of rules executed by u during a GC-segment belongs to the following language:*

$$(\mathbf{R}_I + \varepsilon)(\mathbf{R}_R + \varepsilon)(\mathbf{R}_U)^*(\mathbf{R}_{EB} + \varepsilon)(\mathbf{R}_{EF} + \varepsilon).$$

By Observation 3 and Corollary 2, we obtain the following result.

Theorem 2. *If the number of \mathbf{R}_U executions by any process of GC in any GC-segment is bounded by nb_UN , then the total number of moves in any execution is bounded by $(nb_UN + 4) \cdot (n_{\max CC} + 1) \cdot n$.*

4.4 Causal chains

We now use the notion of *causal chain* defined below to further analyze the number of moves and steps in a GC-segment.

Definition 9 (Causal Chain). *Let GC be a connected component of G . Let v_0 be a process of GC and seg be any GC-segment. A causal chain of seg rooted at v_0 is a non-empty sequence of actions a_1, a_2, \dots, a_k executed in seg such that the action a_1 sets $parent_{v_1}$ to v_0 and for all $2 \leq i \leq k$, the action a_i sets $parent_{v_i}$ to v_{i-1} after the action a_{i-1} but not later than v_{i-1} 's next action.*

Observation 4 *Let GC be a connected component of G , v_0 be a process of GC , and seg be any GC-segment. Let a_1, a_2, \dots, a_k be a causal chain of seg rooted at v_0 . Denote by v_i the process that executes a_i , for all $i \in \{1, \dots, k\}$.*

- For all $1 \leq i \leq k$, a_i consists in the execution of $computePath(v_i)$ (i.e., v_i executes the rule \mathbf{R}_U or \mathbf{R}_R), where v_i is a process of GC .

- Assume a_1 is executed in the step $\gamma \mapsto \gamma'$ of *seg*. Denote by ds_0 the distance value of process v_0 in γ , called the initiating value of the causal chain. For all $1 \leq i \leq k$, a_i sets d_{v_i} to $((ds_0 \oplus w_{v_1}(v_0)) \oplus \dots) \oplus w_{v_i}(v_{i-1})$.

Lemma 8. *Let GC be a connected component of G . Let seg be a segment of GC .*

- All actions in a causal chain of *seg* are executed by different processes of GC .
- Moreover, an execution of $computePath(v)$ by some process v never belongs to any causal chain rooted at v .

Proof. First, by definition, all actions executed in a causal chain of *seg* are executed by processes in GC .

Then, note that any rule \mathbf{R}_U executed by a process v makes the value of d_v decrease.

Assume now, by contradiction, that there exists a process v such that, in some causal chain a_1, a_2, \dots, a_k of *seg*, v is designated as parent in some action a_i executed in step $\gamma_i \mapsto \gamma_{i+1}$ and executes the action a_j in step $\gamma_j \mapsto \gamma_{j+1}$, with $j > i$. v has status C in γ_i , and the value of d_v is strictly larger in γ_{j+1} than in γ_i (Observation 4). So v must execute \mathbf{R}_{EF} between γ_{i+1} and γ_j . Consequently, actions a_i and a_j are executed in two different segments (Lemma 7), a contradiction.

Therefore, all actions in a causal chain are caused by different processes, and a process never executes an action in a causal chain it is the root of. \square

Maximal Causal chains.

Definition 10 (Maximal causal chain). *Let GC be a connected component of G . Let v_0 be a process of GC and let seg be any GC -segment.*

A maximal causal chain of *seg* rooted at v_0 is a causal chain a_1, a_2, \dots, a_k executed in *seg* such that the causal chain is maximal and, either v_0 is a normal root or the action a_1 sets $parent_{v_1}$ to v_0 not later than any action by v_0 in *seg*.

The following lemma adds an additional property to Observation 4 for the specific case of maximal causal chains.

Lemma 9. *Given any connected component GC , any GC -segment seg , and any process $v \in GC$, all maximal causal chains of seg rooted at v have the same initiating value.*

Proof. For the purpose of contradiction, assume that there exist such GC , seg and v such that two maximal causal chains of *seg* rooted at v have different initiating values d_1 and d_2 . At least one of them, say d_1 , must be different from $distRoot_v$. This value d_1 is necessarily the distance value of v at the beginning of *seg*, otherwise v would not be the root of the corresponding maximal causal chain. As a consequence, we must have $d_2 = distRoot_v$.

Since d_1 is the distance value of v at the beginning of *seg*, there must exist an action a executing $beRoot(u)$ in *seg*. By Corollary 2, the action a is an execution of \mathbf{R}_U in the case when $P_updateRoot(v) \wedge st_v = C$ holds. By definition of a GC -segment, the action a is thus executed during the last step of *seg* and thus no maximal causal chains of *seg* (which are never empty by definition) can be rooted at v with initiating value $d_2 = distRoot_v$. This contradiction concludes the proof. \square

Definition 11 ($SI_{seg,v}$). *Let GC be a connected component of G . Let v be a process of GC and let seg be a segment of GC .*

We define $SI_{seg,v}$ as the set of all the distance values obtained after executing an action belonging to the maximal causal chains of *seg* rooted at v .

Lemma 10. *Let GC be a connected component of G . Let v_0 be a process of GC and let seg be a segment of GC . The size of the set SI_{seg,v_0} is bounded by $n_{maxCC}!$.*

Proof. Let us consider a distance value d obtained after executing an action a_i belonging to a maximal causal chain a_1, a_2, \dots, a_k of *seg* rooted at v_0 . Denote by v_i the process that executes a_i , for all $i \in \{1, \dots, k\}$. By Observation 4, we have $d = ((ds_0 \oplus w_{v_1}(v_0)) \oplus \dots) \oplus w_{v_i}(v_{i-1})$, with ds_0 being the initiating value common to all maximal causal chains of *seg* rooted at v_0 . Differently speaking, the value d is fully determined by the sequence of processes v_1, \dots, v_i (v_0 and *seg* being fixed). Moreover, note that all the v_j , $0 \leq j \leq i$ are different processes, by Lemma 8. Therefore, $|SI_{seg,v_0}|$ is bounded by $n_{maxCC}!$. \square

4.5 Move complexity of Scheme

Lemma 11. *Let GC be a connected component of G , $u \in GC$, and seg be a GC -segment. If the size of $SI_{seg,v}$ is bounded by X for any process $v \in GC$, then the number of \mathbf{R}_U moves done by u in seg is bounded by $X \cdot (n_{\max CC} - 1) + 1$.*

Proof. First, assume that $\mathbf{R}_U(u)$ is executed in some step $\gamma \mapsto \gamma'$ of seg and later in some other step $\gamma'' \mapsto \gamma'''$ of seg . By Corollary 2, any sequence of $\mathbf{R}_U(u)$ executions in seg makes the value of d_u decrease. Therefore, all the values of d_u obtained by the \mathbf{R}_U executions done by u are different. By definitions 10 and 11, all these values belong to the set $\bigcup_{v \in GC \setminus \{u\}} SI_{seg,v} \cup \{distRoot(u)\}$, which has size at most $X \cdot (n_{\max CC} - 1) + 1$. \square

By Theorem 2 and Lemma 11, we obtain the following result.

Corollary 3. *If the size of $SI_{seg,v}$ is bounded by X for any connected component GC , any process $v \in GC$, and any GC -segment seg , then the total number of moves during any execution, is bounded by $(X \cdot (n_{\max CC} - 1) + 5) \cdot (n_{\max CC} + 1) \cdot n$.*

Combined with Lemma 10, this corollary already allows us to prove that Scheme always terminates and has a bounded move complexity.

Corollary 4. *Algorithm Scheme is silent self-stabilizing under the distributed unfair daemon and has a bounded move (and step) complexity.*

Let $W_{\max} = \max\{w_u(v) : u \in V \wedge v \in \Gamma(u)\}$. If all weights are strictly positive integers and \oplus is the addition operator, then the size of any $SI_{seg,u}$ is bounded by $W_{\max}(n_{\max CC} - 1)$ for all connected component GC , all GC -segment seg and all process $u \in GC$ because $S_{seg,u} \subseteq [ds_{seg,u} + 1, ds_{seg,u} + W_{\max}(n_{cc} - 1)]$, where $n_{cc} \leq n_{\max CC}$ is the number of processes in GC , and $ds_{seg,u}$ is the common (by Lemma 9) initiating value of the maximal causal chains of seg rooted at u . Hence, we deduce the following theorem from Lemma 1, Theorem 1, and Corollary 3.

Theorem 3. *Algorithm Scheme is silent self-stabilizing under the distributed unfair daemon and, when all weights are strictly positive integers and \oplus is the addition operator, its stabilization time in moves (and steps) is at most $(W_{\max} \cdot (n_{\max CC} - 1)^2 + 5) \cdot (n_{\max CC} + 1) \cdot n$.*

Lemma 12. *Let GC be a connected component of G , $v \in GC$, and seg be GC -segment. If all edges have the same weight, then $|SI_{seg,v}| < n_{\max CC}$.*

Proof. Assume that all edges have the same weight w . According to Observation 4 and Lemma 8, we have $SI_{seg,v} \subset \{ds_{seg,v} \oplus i \cdot w \mid 1 \leq i \leq n_{\max CC} - 1\}$, with $ds_{seg,v}$ being the common (by Lemma 9) initiating value of the maximal causal chains of seg rooted at v . \square

By Corollary 3 and Lemma 12, we obtain the following result.

Corollary 5. *If all edges have the same weight, then the total number of moves (and steps) during any execution, is bounded by $((n_{\max CC} - 1)^2 + 4) \cdot (n_{\max CC} + 1) \cdot n$.*

5 Round Complexity of Scheme

5.1 Normal Configurations.

We first introduce the notion of normal configurations, which will help us to partition the proof on the round complexity of Scheme.

Definition 12 (Normal Process). *A process u is said to be normal if u satisfies the following two conditions:*

1. $st_u \notin \{EB, EF\}$,
2. $\neg P_abnormalRoot(u)$.

Definition 13 (Normal Configuration). Let γ be a configuration of Scheme. γ is said to be normal if every process is normal in γ ; otherwise γ is said to be abnormal.

Observation 5 In a normal configuration of Scheme, only the rules \mathbf{R}_U or \mathbf{R}_R may be enabled on any process.

We first prove that, once a normal configuration is reached, all subsequent configurations will be normal as well.

Lemma 13. Any step from a normal configuration of Scheme reaches a normal configuration of Scheme.

Proof. Let $\gamma \mapsto \gamma'$ be a step such that γ is a normal configuration and let u be a process.

In γ , every process v satisfies $st_v \notin \{EB, EF\}$ and $\neg P_abnormalRoot(v)$. Hence, both $\mathbf{R}_{EB}(u)$ and $\mathbf{R}_{EF}(u)$ are disabled in γ , and consequently $st_u \notin \{EB, EF\}$ still holds in γ' .

Moreover, since u is not an alive abnormal root in γ , Lemma 5 implies that u is not an alive abnormal root in γ' either. Since $st_u \neq EF$ in γ' , we obtain $\neg P_abnormalRoot(u)$ in γ' . \square

5.2 From an Arbitrary Configuration to a Normal Configuration

The first lemma below essentially claims that all processes that are in illegal branches progressively switch to status EB within $n_{\max CC}$ rounds, in order of increasing depth (Definition 5, page 7).

Lemma 14. Let $i \in \mathbb{N}$. From the beginning of round $i+1$, there does not exist any process both in state C and at depth less than i in an illegal branch.

Proof. We prove this lemma by induction on i . The base case ($i = 0$) is vacuum, so we assume that the lemma holds for some integer $i \geq 0$.

From the beginning of round $i+1$, no process can ever choose a parent which is at depth smaller than i in an illegal branch because those processes (if they exist) will never have status C , by induction hypothesis.

Then, let u be a process of status C in an illegal branch at the beginning of round $i+1$. Its depth is thus at least i . By induction hypothesis, each of its ancestor at depth smaller than i has status EB and has at least one child not having status EF . Thus, no such ancestors can execute any rule, and consequently they cannot make the depth of u decreasing to i or smaller. Therefore, no process can take state C at depth smaller or equal to i in an illegal branch from the beginning of round $i+1$.

Consider any process u with status C at depth i in an illegal branch at the beginning of the round $i+1$. By induction hypothesis, u is an abnormal root, or the parent of u is not in state C (i.e., it is in the state EB). During round $i+1$, u will execute rule either \mathbf{R}_{EB} or \mathbf{R}_U and thus either switch to state EB , or join another branch at a depth greater than i , or become a normal root turning its branch to be legal. This concludes the proof of the lemma. \square

Corollary 6. After at most $n_{\max CC}$ rounds, the system is in a configuration from which no process in any illegal branch has status C forever.

Moreover, once such a configuration is reached, each time a process executes a rule other than \mathbf{R}_{EF} , this process is outside any illegal branch forever.

The next lemma essentially claims that once no process in an illegal branch has status C forever, processes in illegal branches progressively switch to status EF within at most $n_{\max CC}$ rounds, in order of decreasing depth.

Lemma 15. Let $i \in \mathbb{N}^*$. From the beginning of round $n_{\max CC} + i$, any process at depth larger than $n_{\max CC} - i$ in an illegal branch has status EF .

Proof. We prove this lemma by induction on i . The base case ($i = 1$) is vacuum (by Observation 2, page 7), so we assume that the lemma holds for some integer $i \geq 1$. At the beginning of round $n_{\max CC} + i$, any process at depth larger than $n_{\max CC} - i$ has the status EF (by induction hypothesis). Therefore, processes with status EB at depth $n_{\max CC} - i$ in an illegal branch can execute the rule \mathbf{R}_{EF} at the beginning of

round $n_{\max\text{CC}} + i$. These processes will thus all execute within round $n_{\max\text{CC}} + i$ (they cannot be neutralized as no children can connect to them) and obtain status EF . We conclude the proof by noticing that, from Corollary 6, once round $n_{\max\text{CC}}$ has terminated, any process in an illegal branch that executes some rule either gets status EF , or will be outside any illegal branch forever. \square

The next lemma essentially claims that after the propagation of status EF in illegal branches, the maximum length of illegal branches progressively decreases until all illegal branches vanish.

Lemma 16. *Let $i \in \mathbb{N}^*$. From the beginning of round $2n_{\max\text{CC}} + i$, there does not exist any process at depth larger than $n_{\max\text{CC}} - i$ in an illegal branch.*

Proof. We prove this lemma by induction on i . The base case ($i = 1$) is vacuum (by Observation 2), so we assume that the lemma holds for some integer $i \geq 1$. By induction hypothesis, at the beginning of round $2n_{\max\text{CC}} + i$, no process is at depth larger than $n_{\max\text{CC}} - i$ in an illegal branch. All processes in an illegal branch have the status EF (by Lemma 15). So, at the beginning of round $2n_{\max\text{CC}} + i$, any abnormal root satisfies the predicate P_reset , and is enabled to execute either \mathbf{R}_I , or \mathbf{R}_R . So, all abnormal roots at the beginning of the round $2n_{\max\text{CC}} + i$ are no more in an illegal branch at the end of this round: the maximal depth of the illegal branches has decreased, since by Corollary 6, no process can join an illegal tree during the round $2n_{\max\text{CC}} + i$. \square

By Lemmas 14-16, we obtain the following result.

Theorem 4. *After at most $3n_{\max\text{CC}}$ rounds, a normal configuration of Scheme is reached.*

5.3 From a Normal Configuration to a Terminal Configuration

From a normal configuration, Algorithm Scheme needs additional rounds to propagate the status C and the correct distances in the components of the graph containing at least one candidate. First, we observe the following fact.

Observation 6 *In a normal configuration of Scheme, all processes in connected components containing no process satisfying canBeRoot are in state I and thus are disabled.*

Let u be a process having the status C in a normal configuration γ . Along any execution from γ , the distance of u cannot increase and u keeps the status C .

From the previous observation, we only need to focus on any connected component GC containing at least one process satisfying canBeRoot .

Let us fix an arbitrary execution ex of Scheme in GC starting from a normal configuration γ . By Corollary 4 (page 13), a terminal configuration is eventually reached after a finite number of steps along ex .

Lemma 17. *Let $\text{ST}_{GC}(i, \text{ex})$ be the set of processes defined by $\{u \in GC \mid u \text{ performs a move along } \text{ex} \text{ after the beginning of the round } i\}$.*

If $|\text{ST}_{GC}(i, \text{ex})| > 0$ then $|\text{ST}_{GC}(i + 1, \text{ex})| < |\text{ST}_{GC}(i, \text{ex})|$.

Proof. By definition, $\text{ST}_{GC}(i + 1, \text{ex}) \subseteq \text{ST}_{GC}(i, \text{ex})$. It is thus sufficient to prove that at least one process of $\text{ST}_{GC}(i, \text{ex})$ is enabled at the start of the i -th round and will do its last action during the i th round of ex .

Let γ_i be the configuration at the start of round i of ex , and let γ_f be the terminal configuration of ex . Let us consider the process $u \in \text{ST}_{GC}(i, \text{ex})$ having the minimum distance d_u in γ_f , denoted by $\text{dmin}(i)$. Along ex from γ_i , any process w' of $\text{ST}_{GC}(i, \text{ex})$ satisfies $\text{dmin}(i) \preceq d_{w'}$ or $st_{w'} = I$ according to the definition of u and to Observation 6.

Case 1. In γ_f , $\text{parent}_u = \perp$.

This means that $P_root(u)$ holds in γ_f . This further implies that, along ex from γ_i , the last action of u consists in executing $\text{beRoot}(u)$. At that time, u satisfies $P_updateRoot(u) \vee st_u = I$. In fact, this must hold already from γ_i , by Lemma 6, page 10) and Observation 6. To summarize, u performs only one action along ex from γ_i , and u is enabled in γ_i and stays enabled until u does an action. This action is thus done during the i th round of ex , and $u \notin \text{ST}_{GC}(i + 1, \text{ex})$, concluding the case.

Case 2. In γ_f , $\text{parent}_u = w$.

By Observation 6, notice that along ex from γ_i , the value $\min_{(v \in \Gamma(u) \wedge \text{st}_v = C)}(d_v \oplus \omega_u(v))$ remains constant, and equal to $\text{dmin}(i)$. Therefore, once u is disable, it stays disable according to $P_nodeImp(u)$ properties (Algorithm 1, page 8). So u is enabled in γ_i and stays enabled until u does an action. Therefore, this action is done during the i -th round of ex . After this action of u , we have $\neg P_updateNode(u) \wedge \neg P_updateRoot(u) \wedge \text{st}_u = C$ (so u is disabled forever). Thus $u \notin \text{ST}_{GC}(i+1, \text{ex})$, concluding the case. □

From the previous lemma, Lemma 13 and Theorem 4, we obtain the following result.

Corollary 7. *A terminal legitimate configuration of any instantiation of Scheme is reached in at most $4n_{\max CC}$ rounds from any configuration.*

6 Instantiations

In this section, we illustrate the versatility of Algorithm Scheme by proposing several instantiations that solve various classical problems. Following the general bound (Corollary 7, page 16), all these instances reach a terminal configuration in at most $4n_{\max CC}$ rounds, starting from an arbitrary one.

6.1 Spanning Forest and Non-Rooted Components Detection

Given an input set of processes rootSet , Algorithm Forest is the instantiation of Scheme with the parameters given in Algorithm 2. Algorithm Forest computes (in a self-stabilizing manner) a spanning forest in each connected component of G containing at least one process of rootSet . The forest consists of trees (of arbitrary topology) rooted at each process of rootSet . Moreover, in any component containing no process of rootSet , the processes eventually detect the absence of root by taking the status I (Isolated).

Algorithm 2: Parameters for any process u in Algorithm Forest

Inputs

- canBeRoot_u is true if and only if $u \in \text{rootSet}$
- pname_u is \perp
- $\omega_u(v) = 1$ for every $v \in \Gamma(u)$

Ordered Magma

- $\text{DistSet} = \mathbb{N}$
- $i1 \oplus i2 = i1 + i2$
- $i1 \prec i2 \equiv (i1 < i2)$
- $\text{distRoot}(u) = 0$

Predicate

- $P_nodeImp(u) \equiv P_updateRoot(u)$
-

Correctness of Forest. By Theorem 1, and Corollary 4 (resp. page 10, 13), Algorithm Forest self-stabilizes to a terminal legitimate configuration that satisfies the following requirements (see Definition 7, page 9).

Observation 7 *In a terminal legitimate configuration of Forest, each process u satisfies one of the following conditions:*

1. $P_root(u)$, i.e., u is a tree-root and $u \in \text{rootSet}$,

2. there is a process of *rootSet* in V_u , $st_u = C$, $parent_u \in \Gamma(u)$, $d_u \geq d_{parent_u} + 1$, and $\neg P_nodeImp(u)$, i.e., $u \notin rootSet$ belongs to a tree rooted at some process of *rootSet* and its neighbor $parent_u$ is its parent in the tree,
3. there is no process of *rootSet* in V_u and $st_u = I$, i.e., u is isolated.

Move Complexity of Forest. Since for every process u , $P_nodeImp(u) \equiv P_updateRoot(u)$, rule \mathbf{R}_U is enabled at most once. Hence, the total number of moves (and steps) during any execution is bounded by $5 \cdot (n_{\max CC} + 1) \cdot n$, by Theorem 2 (page 11).

6.2 Leader Election

Assuming the network is identified, Algorithm LEM is the instantiation of Scheme with the parameters given in Algorithm 3. In each connected component, Algorithm LEM elects the process u (i.e., $P_leader(u)$ holds) of smallest identifier and builds a tree (of arbitrary topology) rooted at u that spans the whole connected component.

Algorithm 3: Parameters for any process u in Algorithm LEM

Inputs

- $canBeRoot_u$ is true for any process
- $pname_u$ is the identifier of u (n.b., $pname_u \in \mathbb{N}$)
- $\omega_u(v) = (\perp, 1)$ for every $v \in \Gamma(u)$

Ordered Magma

- $DistSet = IDs \times \mathbb{N}$; for every $d = (a, b) \in DistSet$, we let $d.id = a$ and $d.h = b$
- $(id1, i1) \oplus (id2, i2) = (id1, i1 + i2)$.
- $(id1, i1) \prec (id2, i2) \equiv (id1 < id2) \vee [(id1 = id2) \wedge (i1 < i2)]$
- $distRoot(u) = (pname_u, 0)$

Predicates

- $P_nodeImp(u) \equiv ((\exists v \in \Gamma(u) \mid st_v = C \wedge d_v.id < d_u.id)) \vee P_updateRoot(u)$
 - $P_leader(u) \equiv P_root(u)$
-

Correctness of LEM. As $canBeRoot$ is true for all processes, we can deduce, from Theorem 1 (page 10) and Definition 7 (page 9), that in a terminal configuration, $st_u = C$ for every process u . So, from Lemma 1 (page 9), Algorithm LEM self-stabilizes to a terminal legitimate configuration that satisfies the following requirements.

Observation 8 *In a terminal legitimate configuration of LEM, each process u satisfies one of the following conditions: (1) $P_root(u)$, or (2) $st_u = C$, $parent_u \in \Gamma(u)$, $d_u \succ d_{parent_u}$.*

Correctness of LEM is proven by the following lemma.

Lemma 18. *In a terminal legitimate configuration of Algorithm LEM, each process u satisfies one of the following conditions:*

1. $P_root(u) (\equiv P_leader(u))$ and u is the process of smallest identifier in V_u , or
2. $st_u = C$, $parent_u \in \Gamma(u)$, $d_u \succ d_{parent_u}$, and $d_u = (pname_{\ell}, -)$ where ℓ is the process of smallest identifier in V_u .

Proof. First, from the previous observation, in a terminal configuration $st_u = C$, for every process u . Then, consider any connected component GC . Assume, by the contradiction, that in a terminal configuration of LEM, we have two processes $u, v \in GC$ such that $d_u.id \neq d_v.id$. Without the loss of generality, assume that u and v are neighbors and $d_u.id > d_v.id$. Then, $P_nodeImp(u)$ holds, and since $st_u = st_v = C$, $\mathbf{R}_U(u)$ is enabled, a contradiction. Hence, all processes of GC agree on the same leader identifier, and by definition of $P_root(u)$, at most one process u can satisfy $P_root(u)$, i.e., $P_leader(u)$.

Assume, then, by the contradiction, that no process of GC that satisfies P_root in the terminal configuration. Let $u \in GC$ such that d_u is minimum in the terminal configuration. By Observation 8, $parent_u \in \Gamma(u)$ and $d_{parent_u} \prec d_u$, contradicting the minimality of d_u . Hence, there is exactly one process ℓ in GC satisfying $P_root(\ell) (\equiv P_leader(\ell))$ in any terminal configuration. Moreover, by Observation 8, in a terminal configuration, $parent$ variables describe a spanning tree rooted at ℓ .

Finally, assume, by the contradiction, that in a terminal configuration, the leader ℓ of GC is not the process of smallest identifier in GC . Let u be the process of smallest identifier in GC . Then, $distRoot(u) = (pname_u, 0) \prec d_u = (pname_\ell, x)$, with $x \in \mathbb{N}$, i.e., $P_updateRoot(u)$. Since $st_u = C$, $\mathbf{R}_U(u)$ is enabled, a contradiction. \square

Move Complexity of LEM. During a GC -segment, a process can only execute \mathbf{R}_U to improve its ID. Since there are $n_{\max CC}$ initial values and $n_{\max CC}$ real IDs in its connected component, the total number of moves (and steps) during any execution is bounded by $(2n_{\max CC} + 4) \cdot (n_{\max CC} + 1) \cdot n$ (Theorem 2, page 11) i.e., $O(n_{\max CC}^2 \cdot n)$.

6.3 Shortest-Path Tree and Non-Rooted Components Detection

Assuming the existence of a unique root r and (strictly) positive integer weights for each edge, Algorithm RSP is the instantiation of Scheme with the parameters given in Algorithm 4. Algorithm RSP computes (in a self-stabilizing manner) a shortest-path tree spanning the connected component of G containing r . Moreover, in any other component, the processes eventually detect the absence of r by taking the status I (Isolated).

Recall that the *weight of a path* is the sum of its edge weights. The *weighted distance* between the processes u and v , denoted by $d(u, v)$, is the minimum weight of a path from u to v . A *shortest path* from u to v is then a path whose weight is $d(u, v)$. A *shortest-path (spanning) tree rooted at r* is a tree rooted at r that spans V_r and such that, for every process u , the unique path from u to r in T is a shortest path from u to r in V_r .

Algorithm 4: Parameters for any process u in Algorithm RSP

Inputs

- $canBeRoot_u$ is false for any process except for $u = r$
- $pname_u$ is \perp
- $\omega_u(v) = \omega_v(u) \in \mathbb{N}^*$, for every $v \in \Gamma(u)$

Ordered Magma

the same as the configuration of Algorithm Forest (Algorithm 2)

Predicate

- $P_nodeImp(u) \equiv P_updateNode(u) \vee P_updateRoot(u)$
-

Correctness of RSP. It is given by the following lemma.

Lemma 19. *In a terminal configuration of Algorithm RSP, r is the unique process satisfying P_root .*

Proof. By definition, r is the unique process satisfying $canBeRoot$. So, only r can satisfy P_root . Assume, by the contradiction, that $\neg P_root(r)$ holds in a terminal configuration γ of RSP. Then, by Definition 7 (page 9) and Theorem 1 (page 10), $st_r = C$ and $d_u \geq d_{parent_u} + 1 > 0$ in γ . So, $st_r = C \wedge P_updateRoot(u)$ holds in γ , *i.e.*, $\mathbf{R}_U(r)$ is enabled in γ , a contradiction. \square

By Lemma 19, Definition 7, and the definition of $P_nodeImp(u)$, we obtain the following result.

Observation 9 *In a legitimate configuration of Algorithm RSP, each process u satisfies one of the following three conditions:*

1. $u = r$ and $P_root(r)$ holds,
2. $u \in V_r \setminus \{r\}$, $st_u = C$, $parent_u \in \Gamma(u)$, and $d_u = d(u, r) = d_{parent_u} + \omega_u(parent_u)$, or
3. $u \notin V_r$ and $st_u = I$.

Move Complexity of RSP. All edges have a positive integer weight, so the total number of moves (and steps) during any execution is bounded by $(W_{max} \cdot (n_{maxCC} - 1)^2 + 5) \cdot (n_{maxCC} + 1) \cdot n$ (Theorem 3, page 13), *i.e.*, $O(n_{maxCC}^3 \cdot n \cdot W_{max})$.

6.4 Leader Election and Breadth-First Search Tree

Assuming the network is identified, Algorithm LEM_BFS is the instantiation of Scheme with the parameters given in Algorithm 5. In each connected component, Algorithm LEM_BFS elects the process u (*i.e.*, $P_leader(u)$ holds) of smallest identifier and builds a breadth-first search (BFS) tree rooted at u that spans the whole connected component.

Recall that the *weight of a path* is the sum of its edge weights (in this case, each edge as weight 1). The *weighted distance* between the processes u and v , denoted by $d(u, v)$, is the minimum weight of a path from u to v . A *shortest path* from u to v is then a path whose weight is $d(u, v)$. When all edges have weight 1, a *BFS spanning tree rooted at u* is a shortest-path (spanning) tree rooted at process u that spans V_u .

Algorithm 5: Parameters for any process u in Algorithm LEM_BFS

Inputs

the same as the configuration of Algorithm LEM (Algorithm 3)

Ordered Magma

the same as the configuration of Algorithm LEM (Algorithm 3)

Predicates

- $P_nodeImp(u) \equiv P_updateNode(u) \vee P_updateRoot(u)$
 - $P_leader(u) \equiv P_root(u)$
-

Correctness of LEM_BFS. Following the same reasoning as for Algorithm LEM and $P_nodeImp(u)$ definition. Algorithm LEM_BFS self-stabilizes to a terminal legitimate configuration that satisfies the following requirements.

Observation 10 *In a terminal legitimate configuration of Algorithm LEM_BFS, each process u satisfies one of the following conditions:*

1. $P_root(u)$ ($\equiv P_leader(u)$) and u is the process of smallest identifier in V_u , or
2. $st_u = C$, $parent_u \in \Gamma(u)$, $d_u = (pname_{\ell}, d(u, \ell)) = d_{parent_u} \oplus (\perp, 1)$, where ℓ is the process of smallest identifier in V_u .

Move Complexity of LEM_BFS. All edges have the same weight, so the total number of moves (and steps) during any execution is bounded by $((n_{\max\text{CC}} - 1)^2 + 5) \cdot (n_{\max\text{CC}} + 1) \cdot n$ (Corollary 5, page 13), *i.e.*, $O(n_{\max\text{CC}}^3 \cdot n)$.

6.5 Depth-First Search Tree and Non-Rooted Components Detection

Assume the existence of a unique root r . Algorithm RDFS is the instantiation of Scheme with the parameters given in Algorithm 6. Algorithm RDFS computes (in a self-stabilizing manner) a depth-first search (DFS) tree spanning the connected component of G containing r . Moreover, in any other component, processes eventually detect the absence of r by taking the status I (Isolated).

Here, the *weight of the arc* (u, v) is $\alpha_u(v)$, the local label of u in $\Gamma(v)$. Let $\mathcal{P} = u_k, u_{k-1}, \dots, u_0 = r$ be a (directed) path from process u_k to the root r . We define the *weight of \mathcal{P}* as the sequence $0, \alpha_1(u_0), \alpha_2(u_1), \dots, \alpha_k(u_{k-1})$. The lexicographical distance from process u to the root r , denoted by $d_{lex}^r(u)$, is the minimum weight of a path from u to r (according to the lexicographical order).

Algorithm 6: Parameters for any process u in Algorithm RDFS

Inputs

- $canBeRoot_u$ is false for any process except for $u = r$
- $pname_u$ is \perp
- $\omega_u(v) = \alpha_u(v) \in \{1 \dots \delta_u\}$ (the local label of u in $\Gamma(v)$), for every $v \in \Gamma(u)$

Ordered Magma

- $DistSet = \{0, \dots, \Delta\}^*$
- $w1 \oplus w2 = w1.w2$ (the concatenation of $w1$ and $w2$)
- \succ is the lexicographic order
- $distRoot(u) = 0$

Predicate

- $P_nodeImp(u) \equiv P_updateNode(u) \vee P_updateRoot(u)$
-

Correctness of RDFS.

Observation 11 *Let T be a tree rooted at r that spans V_r . Following the result of [45], if for every process $u \in V_r$, the weight of the path from u to r in T is equal to $d_{lex}^r(u)$, then T is a (first) DFS spanning tree of V_r .*

Following the same reasoning as for Algorithm RSP, we know that Algorithm RDFS self-stabilizes to a terminal legitimate configuration that satisfies the following requirements.

Observation 12 *In a legitimate configuration of Algorithm RDFS, each process u satisfies one of the following three conditions:*

1. $u = r$ and $P_root(r)$ holds,
2. $u \neq r$, $u \in V_r$, $st_u = C$, $parent_u \in \Gamma(u)$, and $d_u = d_{lex}(u, r) = d_{parent_u}.\omega_u(parent_u)$, or
3. $u \notin V_r$ and $st_u = I$.

Move Complexity of RDFS. For this instance, we cannot apply Theorem 3 (page 13) to obtain a polynomial move complexity. However, by Lemma 10 we have a rough estimation of the move complexity, *i.e.*, at most $n_{\max\text{CC}}!$ moves. We outline that this estimation is coarse-grained, and so can be further refined.

7 Conclusion

We proposed a general scheme, Algorithm Scheme, to compute spanning-tree-like data structures on arbitrary (not necessarily connected) bidirectional networks. Algorithm Scheme is self-stabilizing and silent. It is written in the locally shared memory model with composite atomicity. We proved its correctness under the distributed unfair daemon hypothesis, the weakest scheduling assumption of the model. We also showed that its stabilization time is at most $4n_{\max\text{CC}}$ rounds, where $n_{\max\text{CC}}$ is the maximum number of processes in a connected component. We illustrated the versatility of our approach by proposing several instantiations of Scheme that solve various classical problems. In most of the cases, we exhibited polynomial upper bounds on its stabilization time in steps and process moves for the considered instantiation. For example, assuming the network is identified, we proposed two instances of Scheme for electing a leader in each connected component and building a spanning tree rooted at each leader in a polynomial number of steps (resp. moves). In the first version, the trees are of arbitrary topology, while trees are BFS in the second. Using our scheme, one can easily derive other instances to obtain shortest-path trees for example. Assuming now an input set of roots, we also proposed an instance to compute in a polynomial number of steps (resp. moves) a spanning forest of arbitrary shaped trees, with non-rooted components detection. Again, one can easily enforce this latter construction to obtain BFS or shortest-path forests. Finally, assuming a rooted network, we proposed to compute in a polynomial number of steps (resp. moves) a shortest-path spanning tree construction, with non-rooted components detection. Again, BFS or arbitrary tree constructions can be easily derived from these latter instances. Notice that, for many of these latter problems, there was, until now, no solution in the literature where a polynomial step complexity upper bound was proven.

References

1. Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.
2. Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, Swiss Federal Institute of Technology (EPFL), 2003.
3. Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.
4. G Tel. *Introduction to distributed algorithms*. Cambridge University Press, Cambridge, UK, Second edition 2001.
5. Mohamed G. Gouda and Ted Herman. Adaptive programming. *IEEE Trans. Software Eng.*, 17(9):911–921, 1991.
6. Shlomi Dolev. *Self-stabilization*. MIT Press, March 2000.
7. Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *5th International Workshop on Self-Stabilizing Systems, (WSS 2001)*, Springer LNCS 2194, pages 19–34, 2001.
8. Ajoy Kumar Datta, Shivashankar Gurumurthy, Franck Petit, and Vincent Villain. Self-stabilizing network orientation algorithms in arbitrary rooted networks. *Stud. Inform. Univ.*, 1(1):1–22, 2001.
9. A Arora, MG Gouda, and T Herman. Composite routing protocols. In *the 2nd IEEE Symposium on Parallel and Distributed Processing (SPDP'90)*, pages 70–78, 1990.
10. L. Blin, M. Potop-Butucaru, S. Rovedakis, and S. Tixeuil. Loop-free super-stabilizing spanning tree construction. In *the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer LNCS 6366, pages 50–64, 2010.
11. Ajoy Kumar Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and Yvan Rivierre. Competitive self-stabilizing k-clustering. *Theor. Comput. Sci.*, 626:110–133, 2016.
12. Ajoy Kumar Datta, Lawrence L. Larmore, Stéphane Devismes, Karel Heurtefeux, and Yvan Rivierre. Self-stabilizing small k-dominating sets. *IJNC*, 3(1):116–136, 2013.
13. Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2014)*, Springer LNCS 8756, pages 18–32, 2014.
14. Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing leader election in polynomial steps. *Information and Computation, special issue of SSS 2014*, 2016. To appear.
15. Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. *J. Parallel Distrib. Comput.*, 81-82:11–23, 2015.

16. Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An $o(n)$ -time self-stabilizing leader election algorithm. *jpdc*, 71(11):1532–1544, 2011.
17. Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011.
18. Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks. In *the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'14)*, Springer LNCS 8736, pages 120–134, 2014.
19. Stéphane Devismes and Colette Johnen. Silent self-stabilizing {BFS} tree algorithms revisited. *Journal of Parallel and Distributed Computing*, 97:11 – 23, 2016.
20. Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.
21. Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks - extended version. Technical report, LaBRI, CNRS UMR 5800, 2016.
22. Alain Cournier. A new polynomial silent stabilizing spanning-tree construction algorithm. In *International Colloquium on Structural Information and Communication Complexity*, pages 141–153. Springer, 2009.
23. Adrian Kosowski and Lukasz Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In *6th International Conference Parallel Processing and Applied Mathematics, (PPAM'05)*, Springer LNCS 3911, pages 75–82, 2005.
24. Stéphane Devismes, David Ilcinkas, and Colette Johnen. Self-stabilizing disconnected components detection and rooted shortest-path tree maintenance in polynomial steps. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, volume 70 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
25. J. A. Cobb and C. T. Huang. Stabilization of Maximal-Metric Routing without Knowledge of Network Size. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 306–311, December 2009.
26. Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.
27. Paolo Boldi and Sebastiano Vigna. Universal dynamic synchronous self-stabilization. *Distributed Computing*, 15(3):137–153, July 2002.
28. Alain Cournier, Ajoy Kumar Datta, Stéphane Devismes, Franck Petit, and Vincent Villain. The expressive power of snap-stabilization. *Theor. Comput. Sci.*, 626:40–66, 2016.
29. Emmanuel Godard. Snap-Stabilizing Tasks in Anonymous Networks. In *Stabilization, Safety, and Security of Distributed Systems (SSS'16)*, Lecture Notes in Computer Science, pages 170–184. Springer, Cham, November 2016.
30. Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, 2001.
31. Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10):498–514, 2006.
32. Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.
33. Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *TAAAS*, 4(1):6:1–6:27, 2009.
34. Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), 2009.
35. Alain Cournier, Stephane Rovedakis, and Vincent Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. In *the 15th International Conference on Principles of Distributed Systems (OPODIS'11)*, Springer LNCS 7109, pages 159–174, 2011.
36. Cournier, Alain. A lower bound for the $Max + 1$ algorithm. <https://home.mis.u-picardie.fr/~cournier/MaxPlusUn.pdf>. Online; accessed 11 February 2009.
37. Alain Cournier, Stéphane Devismes, Franck Petit, and Vincent Villain. Snap-stabilizing depth-first search on arbitrary networks. *The Computer Journal*, 49(3):268–280, 2006.
38. Alain Cournier, Stéphane Devismes, and Vincent Villain. A snap-stabilizing dfs with a lower space requirement. In *Symposium on Self-Stabilizing Systems*, pages 33–47. Springer, 2005.
39. M Sloman and J Kramer. *Distributed systems and computer networks*. Prentice Hall, 1987.
40. S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only Read/Write atomicity. *Distributed Computing*, 7(1):3–16, 1993.
41. Alberto Leon-Garcia and Indra Widjaja. *Communication Networks*. McGraw-Hill, Inc., New York, NY, USA, 2 edition, 2004.

42. Lélia Blin, Alain Cournier, and Vincent Villain. An improved snap-stabilizing PIF algorithm. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003*, volume 2704 of *Lecture Notes in Computer Science*, pages 199–214, San Francisco, CA, USA, June 24–25 2003. Springer.
43. Ernest J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.
44. Adrian Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.
45. Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Inf. Process. Lett.*, 49(6):297–301, 1994.