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

Stéphane Devismes, David Ilcinkas, Colette Johnen

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

Stéphane Devismes David Ilcinkas Colette Johnen

December 19, 2017

Abstract

We propose a general scheme, 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 $O(n_{\texttt{maxCC}})$ rounds, where $n_{\texttt{maxCC}}$ is the maximum number of processes in a connected component. We also exhibit polynomial upper bounds on its stabilization time in steps that depend on the considered precise problem. We illustrate the versatility of our approach by proposing several instantiations of Scheme 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

*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).

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 schemes 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. Interestingly, self-stabilizing algorithms designed under this assumption are easier to compose [12]. Moreover, the *stabilization time* can also be bounded in terms of steps 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. Moreover, step complexity captures the amount of computations an algorithm needs to recover a correct behavior. Indeed the number of moves, *i.e.*, the number local state updates, and the number 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 step complexity 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 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, 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 $4n_{\texttt{maxCC}} - 1$ rounds, where $n_{\texttt{maxCC}}$ the maximum number of processes in a connected component.

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

Moreover, its stabilization time in steps is polynomial. Precisely, we exhibit polynomial upper bounds on its stabilization time in steps that depend on the particular problems we consider.

To illustrate the versatility of our approach, we propose five instantiations of Scheme solving classical spanning-tree-like problems. Assuming the network is identified (*i.e.*, nodes have distinct IDs), we propose two instantiations of Scheme, stabilizing in $O({n_{\texttt{maxCC}}}^3 \cdot n)$ steps, for electing a leader in each connected component and building a spanning tree rooted at each leader. In one version, the trees are of arbitrary topology, while trees are BFS in the other. 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. This instance stabilizes in $O(n_{\texttt{maxCC}} \cdot n)$ steps. Finally, assuming a rooted network, we propose shortest-path and DFS spanning tree constructions, with non-rooted components detection, that stabilize in $O({n_{\texttt{maxCC}}}^3 \cdot n \cdot \texttt{W}_{\max})$ and $O(\Delta \cdot {n_{\texttt{maxCC}}}^3 \cdot n)$ ($\texttt{W}_{\max}$ is the maximum weight of an edge and Δ is the maximum degree of the network), respectively. From these various examples, one can easily derive other silent self-stabilizing spanning-tree-like constructions.

This work is inspired by both [22] and [23]. In the former, a generic self-stabilizing algorithm is presented which constructs a spanning tree rooted at some designated node and maximizing some given metric. This algorithm is however only proven for the restricted centralized weakly-fair daemon, for a single designated node, and for connected networks only. [23] on the other hand operates under the distributed unfair daemon, is efficient both in terms of rounds and steps, tolerates disconnections, but it is restricted to the case of the shortest-path tree. Generalizing these two works 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). The structure of the paper will be explained in more details at the end of the introduction.

Related Work. 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 [24, 25, 26, 27]. 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 [24] and [25], 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 [26, 27] assuming the locally shared memory model with composite atomicity and a distributed unfair daemon.

In [28, 29], 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 [28], they consider the locally shared memory model with composite atomicity assuming a distributed unfair daemon, while in [29], 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 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* [30] 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 complexity is given.

Efficient and general schemes for snap-stabilizing waves in arbitrary connected and rooted networks are tackled in [31]. 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 [32, 33]. The algorithm in [32] is not silent and has a stabilization time in $O(\Delta \cdot n^3)$ steps. The silent algorithm given in [33] has a stabilization time $O(D^2)$ rounds and $O(n^6)$ steps. In [23], we propose a silent self-stabilizing disconnected components detection and rooted shortest-path tree maintenance algorithm that stabilizes in an arbitrary rooted network within a linear number of rounds and a polynomial number of steps. Silent self-stabilizing algorithms that construct spanning trees of arbitrary topologies in arbitrary connected and rooted networks are given in [34, 35]. The solution proposed in [34] stabilizes in at most $4 \cdot n$ rounds and $5 \cdot n^2$ steps, while the algorithm given in [35] stabilizes in $n \cdot D$ steps (its round complexity is not analyzed).

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), [36, 37] (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 [36, 37] 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 [36], 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 steps are given in Section 4, whereas an analysis of the stabilization time in rounds is proposed in Section 5. Five instances of Scheme with their specific complexity analysis 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 $n_{\texttt{maxCC}}$ the maximum number of processes in a connected component of G. By definition, $n_{\texttt{maxCC}} \leq n$.

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 [38]. 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', and so on. The transition from γ to γ' is called a *step*. 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 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* [39] 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 $e = \gamma_0, \gamma_1, \cdots$ is the minimal prefix $e' = \gamma_0, \cdots, \gamma_j$, 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 $\gamma_j, \gamma_{j+1}, \cdots$ 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 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 semigroup. The definition of $(DistSet, \oplus, \prec)$ is problem dependent and, if necessary (*i.e.*, if the predicate $P\_nodeImp(u)$[2] holds at some process u), the weight of the trees will be minimized using this ordered semigroup and the distance value that a tree root should have, given by the constant parameter $distRoot(u)$ (which is again problem dependent).

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:

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

[2]The definition of $P\_nodeImp(u)$ is problem dependent, however we require that $P\_nodeImp(u) \Rightarrow P\_updateNode(u)$, as explained later.

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. Precisely, $st_u = C$ (resp. $st_u = I$) means that u believes that its connected component contains at least one candidate root (resp. no candidate). The meaning of status EB and EF will be further detailed in Subsection 3.4.

$parent_u \in \{\perp\} \cup Lbl$**:** If V_u contains a candidate, then in a terminal configuration, u is either a tree root and $parent_u = \perp$, or $parent_u$ belongs to $\Gamma(u)$, *i.e.*, $parent_u$ designates a neighbor of u, referred to as its *parent*.

$d_u \in DistSet$. 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 Normal Execution

Consider any configuration, where for every process u, if $canBeRoot_u$, then $st_u = C$ and $d_u = distRoot(u)$, otherwise $st_u = I$. Refer to such a configuration as a *normal initial configuration*. Each configuration reachable from a *normal initial configuration* is called a *normal configuration*, otherwise it is an *abnormal configuration* (see Definition 9, page 12, for the formal definition). Then, starting from a normal initial configuration, all processes that belong to a connected component containing no candidate are disabled forever. Focus now on a connected component GC where at least one process is candidate. All candidates have status C forever. Then, process u of status I that is neighbor of a process of status C is enabled to execute rule $\mathbf{R_{RN}}$: it eventually executes $\mathbf{R_{RN}}(u)$ to join a tree rooted at some candidate by choosing as parent a neighbor v of status C that minimize its distance value. Using this rule, it also switches its status to C and sets d_u to $d_v \oplus \omega_u(v)$. Executions of rule $\mathbf{R_{RN}}$ are asynchronously propagated in GC until all processes of GC have status C. In parallel, rules $\mathbf{R_{UN}}$ 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 ($P\_nodeImp(u) \Rightarrow P\_updateNode(u)$) 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 sufficently good parent in its neighborhood. Hence, eventually the system reaches a terminal configuration, where a specific spanning forest (maybe a single spanning tree) is (distributedly) defined in connected components containing at least one candidate, while all processes are isolated in other components.

3.4 Error Correction

Assume now that the system is in an abnormal configuration. In this case, we have two kinds of inconsistency, depending on whether or not the local error correction may impact the consistency of the neighbors of the faulty process.

Let us first consider the following two (simple) cases, where a faulty process u can be corrected without impacting the consistency of its neighbors:

1. The predicate $canBeRoot_u \wedge st_u = I$ holds: now, from the previous explanation, a candidate should have status C. In this case, u satisfies $P\_guardRR(u)$ and corrects its state by executing either $\mathbf{R_{RR}}(u)$, or $\mathbf{R_{RN}}(u)$.

2. The predicate $st_u = C \wedge canBeRoot_u \wedge distRoot(u) \prec d_u$ holds: now, from the previous explanation, a candidate u should satisfy $d_u \preceq distRoot(u)$. In this case, u satisfies $P\_updateRoot(u)$ and corrects its state by executing either $\mathbf{R_{UN}}(u)$, or $\mathbf{R_{UR}}(u)$.

Other inconsistencies are detected using predicate $P\_abnormalRoot$. We call *abnormal root* any process u satisfying $P\_abnormalRoot(u)$. Informally (see Subsection 4.1, page 9, 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.

The management of these inconsistencies is more intricate since simply resetting the state of an abnormal root may propagate the inconsistency to some of its neighbors (its children, actually).

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 neighbors (*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 $\mathcal{P} = u_1, \ldots, u_k$ such that $\forall i, 1 \leq i < k$, $u_{i+1} \in Children(u_i)$. $\mathcal{P}$ is acyclic. If u_1 is either a normal or an abnormal root, then $\mathcal{P}$ is called a *branch* rooted at u_1. 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*.

To recover a normal configuration, it is then 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 $\mathbf{R_{UN}}(u)$, then it does so and T becomes a subtree of T'. Otherwise, T has to be 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.) 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 $\mathbf{R_{UN}}(v)$, and (2) no process w can join the tree by executing $\mathbf{R_{RN}}(w)$ or $\mathbf{R_{UN}}(w)$. Once frozen, the tree can be safely deleted from its root to its leaves.

The freezing mechanism (inspired from [40]) 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 states are actually used to perform a "Propagation of Information with Feedback" [41, 42] 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 is two possibilities for the deletion depending on whether or not the process u to be deleted has a neighbor with status C. If u has a neighbor with status C, the rule $\mathbf{R_{RN}}(u)$ 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 has no neighbor with status C, the rule $\mathbf{R_I}(u)$ is executed: if u is a candidate, it becomes a normal root by executing $beRoot(u)$, otherwise u becomes isolated and may 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 $\mathbf{R_{RN}}(v)$ 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 should also be 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 abnormal alive tree, it is in that situation since the initial configuration (see Lemma 1, page 11). The polynomial step complexity mainly relies on this strong property.

4 Correctness and Step Complexity of Scheme

4.1 Definitions

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

4.1.1 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 $Children(parent_v)$).*

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

- $P\_root(u) \equiv canBeRoot_u \wedge st_u = C \wedge parent_u = \perp \wedge d_u = distRoot(u)$
- $P\_abnormalRoot(u) \equiv$
  $\neg P\_root(u) \wedge\ st_u \neq I \wedge\ [parent_u \notin \Gamma(u) \vee$
  $st_{parent_u} = I \vee d_u \prec d_{parent_u} \oplus \omega_u(parent_u) \vee$
  $(st_u \neq st_{parent_u} \wedge st_{parent_u} \neq EB)]$
- $P\_reset(u) \equiv st_u = EF \wedge P\_abnormalRoot(u)$
- $P\_guardRR(u) \equiv canBeRoot_u \wedge st_u = I$
- $P\_updateNode(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C \wedge d_v \oplus \omega_u(v) \prec d_u)$
- $P\_updateRoot(u) \equiv canBeRoot_u \wedge distRoot(u) \prec d_u$
- $P\_nodeImp(u)$ is problem dependent
  (however, if $P\_nodeImp(u)$, then $P\_updateNode(u)$)

Macros

- $beRoot(u)$: $st_u := C$; $parent_u := \perp$; $d_u := distRoot(u)$;
- $computePath(u)$:
  $st_u := C$;
  $parent_u := \text{argmin}_{(v \in \Gamma(u)\ \wedge\ st_v = C)}(d_v\ \oplus\ \omega_u(v))$;
  $d_u := d_{parent_u} \oplus \omega_u(parent_u)$;
  if $P\_updateRoot(u)$ **then** $beRoot(u)$;
- $reset(u)$: **if** $canBeRoot_u$ **then** $beRoot(u)$; **else** $st_u := I$;

Rules

$\mathbf{R_{UN}}(u)$:	$st_u = C \wedge P\_nodeImp(u)$	$\rightarrow$	$computePath(u)$;
$\mathbf{R_{UR}}(u)$:	$st_u = C \wedge \neg P\_nodeImp(u) \wedge P\_updateRoot(u)$	$\rightarrow$	$beRoot(u)$
$\mathbf{R_{EB}}(u)$:	$st_u = C \wedge \neg P\_nodeImp(u) \wedge \neg P\_updateRoot(u) \wedge$ $(P\_abnormalRoot(u) \vee st_{parent_u} = EB)$	$\rightarrow$	$st_u := EB$;
$\mathbf{R_{EF}}(u)$:	$st_u = EB \wedge (\forall v \in Children(u) \mid st_v = EF)$	$\rightarrow$	$st_u := EF$;
$\mathbf{R_{I}}(u)$:	$P\_reset(u) \wedge (\forall v \in \Gamma(u) \mid st_v \neq C)$	$\rightarrow$	$reset(u)$;
$\mathbf{R_{RN}}(u)$:	$(P\_reset(u) \vee st_u = I) \wedge (\exists v \in \Gamma(u) \mid st_v = C)$	$\rightarrow$	$computePath(u)$;
$\mathbf{R_{RR}}(u)$:	$P\_guardRR(u) \wedge (\forall v \in \Gamma(u) \mid st_v \neq C)$	$\rightarrow$	$beRoot(u)$;

Definition 5 (Branch) *A* branch *is a sequence of processes $v_1, \cdots, v_k$, for some integer $k \geq 1$, such that v_1 is a normal or an abnormal root and, for every $1 \leq i < k$, we have $v_{i+1} \in Children(v_i)$. The process v_i is said to be at* depth *i and $v_i, \cdots, v_k$ is called a* sub-branch. *If v_1 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}. 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).*

Lemma 1 *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'.

Consider now the case where u has status either C or EB in γ', and consider the following two subcases:

$parent_u = \perp$ **in** γ'**:** In this case, either u executes $beRoot(u)$ in $\gamma \mapsto \gamma'$, or u executes no action modifying $parent_u$ in $\gamma \mapsto \gamma'$ and thus $parent_u = \perp$ already holds in γ.

In the former case, $canBeRoot_u$ is true and so $P\_root(u)$ is true in γ', which, in turn, implies $\neg P\_abnormalRoot(u)$ in γ'. Consequently, u is still not an alive abnormal root in γ'.

In the latter case, $st_u \in \{C, EB, EF\}$ in γ. If $st_u = EF$ in γ, then $st_u \in \{EF, I\}$ in γ' and u is not an alive abnormal root in γ'. Otherwise, we have that $\neg P\_abnormalRoot(u)$ holds in γ. Now, given that $parent_u = \perp$ in γ, we have $P\_root(u)$ in γ. As u neither executes any action modifying $parent_u$ nor executes $beRoot(u)$ in $\gamma \mapsto \gamma'$, u does not execute any action in $\gamma \mapsto \gamma'$ and thus $P\_root(u)$ still holds in γ'. This implies that $\neg P\_abnormalRoot(u)$ still holds in γ', and so u is still not an alive abnormal root in γ'.

$parent_u \neq \perp$ **in** γ'**:** Let v be the process such that $parent_u = v$ in γ'.

Consider first the case where u has status EB in γ'. The only rule u can execute in $\gamma \mapsto \gamma'$ is $\mathbf{R_{EB}}$ and thus $st_u \in \{C, EB\}$ in γ. Whether u executes $\mathbf{R_{EB}}$ or not, $parent_u$ is also v in γ. Since u is not an alive abnormal root in γ, and thus not an abnormal root either, v necessarily has status EB in γ in either case. Moreover, u belongs to $Children(v)$ in γ. So, v is not enabled in γ and $u \in Children(v)$ remains true in γ'. Hence, we can conclude that u is still not an alive abnormal root in γ'.

Let us study the other case, *i.e.*, u has status C in γ'. During $\gamma \mapsto \gamma'$, the only rules that u may execute are $\mathbf{R_{UN}}$ or $\mathbf{R_{RN}}$. If u executes $\mathbf{R_{UN}}$ or $\mathbf{R_{RN}}$, then we have $st_v = C$ in γ (because it is a requirement to execute any of these rules) and consequently, the only rules that v may execute in $\gamma \mapsto \gamma'$ are $\mathbf{R_{UN}}$, $\mathbf{R_{UR}}$, or $\mathbf{R_{EB}}$. Otherwise (if u does not execute any action in $\gamma \mapsto \gamma'$), $parent_u = v$ and $st_u = C$ already hold in γ. Then, in such a case, u not being an alive abnormal root and $st_u = C$ in γ implies that $\neg P\_abnormalRoot(u)$ and then both $u \in Children(v)$ and $st_v \in \{C, EB\}$ in γ, further implying that the only rules that v may execute in $\gamma \mapsto \gamma'$ are $\mathbf{R_{UN}}$, $\mathbf{R_{UR}}$, or $\mathbf{R_{EB}}$. Thus, in either case, during $\gamma \mapsto \gamma'$, v either takes the status EB, decreases its distance value, or does not change the value of its variables. Consequently, u belongs to $Children(v)$ in γ', which prevents u from being an alive abnormal root in γ'.

□

4.1.2 Legitimate Configuration.

Definition 6 (Legitimate State) *A process u is said to be in a* legitimate state *in* 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, $st_u = C$, $parent_u \in \Gamma(u)$, $d_u \succeq d_{parent_u} \oplus \omega_u(parent_u)$, $\neg P\_nodeImp(u)$, *and* $\neg P\_updateRoot(u)$, *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.*

Lemma 2 *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) $st_v = C$, $parent_v \in \Gamma(v)$, $d_v \succeq d_{parent_v} \oplus \omega_v(parent_v)$, $\neg P\_nodeImp(v)$, and $\neg P\_updateRoot(v)$. This in particular means that $st_v = C$, for every $v \in V_u$. Hence, $\mathbf{R_{EF}}(v)$, $\mathbf{R_{I}}(v)$, $\mathbf{R_{RN}}(v)$, and $\mathbf{R_{RR}}(v)$ are all disabled at every $v \in V_u$ in γ. $\neg P\_nodeImp(v) \wedge (P\_root(v) \vee \neg P\_updateRoot(v))$ then implies that $\mathbf{R_{UN}}(v)$ and $\mathbf{R_{UR}}(v)$ are 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. □

4.1.3 Normal Configuration.

Definition 8 (Normal Process) *A process u is said to be* normal *if u satisfies the following three conditions:*

1. $st_u \notin \{EB, EF\}$,
2. $canBeRoot_u \Rightarrow st_u \neq I \wedge d_u \preceq distRoot(u)$ *(i.e.,* $\neg P\_guardRR(u) \wedge \neg P\_updateRoot(u)$*), and*
3. $\neg P\_abnormalRoot(u)$.

Definition 9 (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 3 *In a normal configuration of* Scheme, *only the rules $\mathbf{R_{UN}}$ or $\mathbf{R_{RN}}$ may be enabled on any process.*

In the following, we say that a predicate P is *closed* if for all steps $\gamma \mapsto \gamma'$, P in $\gamma \Rightarrow P$ in γ'.

Lemma 3 *For every process u, $\neg P\_guardRR(u)$ is closed.*

Proof. Let $\gamma \mapsto \gamma'$ be a step such that $\neg P\_guardRR(u) = \neg canBeRoot_u \vee st_u \neq I$ in γ for some process u.

If $\neg canBeRoot_u$ holds in γ, then $\neg canBeRoot_u$ also holds in γ' ($canBeRoot_u$ is a constant) and so $\neg P\_guardRR(u)$ still holds in γ'.

Otherwise, $canBeRoot_u$ and $st_u \neq I$ in γ. Now, $canBeRoot_u$ implies that if $\mathbf{R_I}(u)$ (the only rule that may set st_u to I) is executed in $\gamma \mapsto \gamma'$, then $st_u \neq I$ still holds in γ' and so $\neg P\_guardRR(u)$ remains true in γ'. □

Lemma 4 *For every process u, $\neg P\_updateRoot(u)$ is closed.*

Proof. Let $\gamma \mapsto \gamma'$ be a step such that $\neg P\_updateRoot(u) = \neg canBeRoot_u \vee distRoot(u) \succeq d_u$ in γ for some process u. Then, $\neg P\_updateRoot(u)$ in γ' since $canBeRoot_u$ is constant and every rule that updates d_u ensures that $distRoot(u) \succeq d_u$ (see, in particular, the last line of macro $computePath(u)$). □

Lemma 5 *For every process u, $\neg P\_updateRoot(u) \vee st_u \neq C$ is closed.*

Proof. Let $\gamma \mapsto \gamma'$ be a step such that $\neg P\_updateRoot(u) \vee st_u \neq C$ in γ for some process u.

If $\neg P\_updateRoot(u)$ holds in γ, then $\neg P\_updateRoot(u) \vee st_u \neq C$ still holds in γ' by Lemma 4.

Otherwise, $P\_updateRoot(u) \wedge st_u \neq C$ holds in γ. If u does not move in $\gamma \mapsto \gamma'$, then $st_u \neq C$ remains true in γ' and, consequently, $(\neg P\_updateRoot(u) \vee st_u \neq C)$ still holds in γ'. Otherwise, u executes either $\mathbf{R_{EF}}(u)$, $\mathbf{R_I}(u)$, $\mathbf{R_{RN}}(u)$, or $\mathbf{R_{RR}}(u)$. If u executes $\mathbf{R_{EF}}(u)$, then $st_u = EF$ in γ' and, consequently, $(\neg P\_updateRoot(u) \vee st_u \neq C)$ still holds in γ'. If u executes $\mathbf{R_I}(u)$, then either $st_u = I$, or $P\_root(u)$, which implies $\neg P\_updateRoot(u)$, holds in γ', consequently, $(\neg P\_updateRoot(u) \vee st_u \neq C)$ still holds in γ'. Finally, if u executes $\mathbf{R_{RN}}(u)$ or $\mathbf{R_{RR}}(u)$, then $\neg P\_updateRoot(u)$, and so, $(\neg P\_updateRoot(u) \vee st_u \neq C)$, holds in γ'.

Hence, in all cases, $P\_updateRoot(u) \wedge st_u \neq C$ holds in γ' and we are done. □

Theorem 1 *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'.

Then, by Lemmas 3 and 4, $canBeRoot_u \Rightarrow st_u \neq I \wedge d_u \succeq distRoot(u)$ (*i.e.*, $\neg P\_guardRR(u) \wedge \neg P\_updateRoot(u)$) holds in γ'.

Finally, since u is not an alive abnormal root in γ, Lemma 1 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'. □

Theorem 2 *Let γ be a normal configuration of* Scheme. *In γ, all processes in connected components containing no process satisfying canBeRoot are isolated.*

Proof. Let GC be a component in which no process satisfies $canBeRoot$. Consider, by contradiction, that there exists a process u in GC such that $st(u) \neq I$ in γ. By definition of normal configuration, u has status C and $\neg P\_abnormalRoot(u)$ in γ. Without loss of generality, assume that u is the process of GC with status C having the smallest distance value d_u in γ. Then, by Observation 1, $P\_root(u)$ holds in γ, which in turn implies $canBeRoot_u$, a contradiction. □

4.2 Partial Correctness

Lemma 6 *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 the 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 $\mathbf{R_{UN}}(v)$, $\mathbf{R_{UR}}(v)$, 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 the 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 $\mathbf{R_{RN}}(u)$ is enabled, a contradiction. □

Lemma 7 *Any terminal configuration of* Scheme *is a normal configuration.*

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

- $st_u \notin \{EB, EF\}$ in γ, by Lemma 6.
- Assume $canBeRoot_u$ holds in γ. Then $st_u \neq I$ in γ because, otherwise, either $\mathbf{R_{RN}}(u)$ or $\mathbf{R_{RR}}(u)$ is enabled in γ. Moreover, $st_u \neq I$ in γ implies that $st_u = C$ in γ, by Lemma 6. Consequently, $d_u \preceq distRoot(u)$ in γ because otherwise either $\mathbf{R_{UN}}(u)$ or $\mathbf{R_{UR}}(u)$ would be enabled in γ.
- $\neg P\_abnormalRoot(u)$ because, otherwise, $st_u = C$ by Lemma 6 and either $\mathbf{R_{UN}}(u)$, $\mathbf{R_{UR}}(u)$, or $\mathbf{R_{EB}}(u)$ would be enabled.

Hence, γ is normal. □

Theorem 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:*

- $st_u = C$,
- $\neg P\_nodeImp(u)$,
- $\neg P\_updateRoot(u)$, *and*
- $P\_root(u)$ *or* $parent_u \in \Gamma(u) \wedge d_u \succeq d_{parent_u} \oplus w_u(parent_u)$.

Proof. Let v be a process of V_u such that $canBeRoot_v$ in γ. In γ, $st_v = C$, by Lemma 7.

Assume then 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 6, whereas at least one process, *e.g.*, v, of V_u has status C). Then, $\mathbf{R_{RN}}(w)$ 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, $\neg P\_nodeImp(u)$, and $\neg P\_updateRoot(u)$ hold in γ (otherwise, either $\mathbf{R_{UN}}(u)$ or $\mathbf{R_{UR}}(u)$ would be enabled).

In γ, u satisfies $\neg P\_abnormalRoot(u)$ (by Lemma 7). So, as $st_u = C$ in γ, we can conclude by Obseravtion 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. □

By Theorems 2, 3, and Lemma 7, we obtain the following result.

Corollary 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 step complexity of any instance of Scheme.

4.3 GC-segment

Let GC be a connected component of G and γ be a configuration. Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that $P\_abnormalRoot(u) \wedge st_u \neq EF$ (u is an abnormal alive root) or $P\_guardRR(u)$ or $P\_updateRoot(u) \wedge st_u = C$ in γ.

By Lemmas 1, 3, and 5, we obtain the following result.

Corollary 2 *For every step $\gamma \mapsto \gamma'$, $SL(\gamma', GC) \subseteq SL(\gamma, GC)$.*

Based on Corollary 2, we can use the notion of *GC-segment* defined below to bound the total number of steps in an execution.

Definition 10 (*GC*-Segment) *Let $e = \gamma_0, \gamma_1, \cdots$ 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 $\gamma_0, \cdots, \gamma_{i+1}$. The* second GC-segment *of e is the first GC-segment of the suffix $\gamma_{i+1}, \gamma_{i+2}, \cdots$, and so forth.*

By Corollary 2, we have

Observation 4 *Let GC be a connected component of G. For every execution e of* Scheme, *e contains at most $n_{maxCC} + 1$ GC-segments, because $|SL(\gamma_i, GC)| \leq n_{maxCC}$ by definition.*

Lemma 8 *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 1) of any branch in γ_1 containing v. 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 $\mathbf{R_{RN}}$ 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 2 and Definition 10. □

Lemma 9 *Let GC be a connected component of G and u be any process of GC. If during the step $\gamma \mapsto \gamma'$, u executes the rule $\mathbf{R_{RR}}$, then $\gamma \mapsto \gamma'$ is the last step of the current GC-segment.*

Proof. In γ, $P\_guardRR(u)$ holds, so $u \in SL(\gamma, GC)$. Now, the execution of $beRoot(u)$ implies that $P\_root(u)$ holds in γ', which in turn implies that $u \notin SL(\gamma', GC)$, hence we are done by Corollary 2 and Definition 10. □

Lemma 10 *Let GC be a connected component of G and u be any process of GC. If during the step $\gamma \mapsto \gamma'$, u executes the rule $\mathbf{R_{UR}}$, then $\gamma \mapsto \gamma'$ is the last step of the current GC-segment.*

Proof. In γ, $P\_updateRoot(u) \wedge st_u = C$ holds, so $u \in SL(\gamma, GC)$. Now, the execution of $beRoot(u)$ implies that $P\_root(u)$ holds in γ', which in turn implies that $u \notin SL(\gamma', GC)$, hence we are done by Corollary 2 and Definition 10. □

By Lemmas 8-10, we obtain the following result.

Corollary 3 *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:*
$[(\mathbf{R_I} + \varepsilon)(\mathbf{R_{RN}} + \varepsilon)(\mathbf{R_{UN}})^*(\mathbf{R_{EB}} + \varepsilon)\mathbf{R_{EF}}] + \mathbf{R_{UR}} + \mathbf{R_{RR}} + \varepsilon)$.

By Observation 4 and Corollary 3, we obtain the following result.

Theorem 4 *If the number of $\mathbf{R_{UN}}$ executions by any process of GC in any GC-segment is bounded by nb_UN, then the total number of steps in any execution is bounded by $(nb\_UN{+}4){\cdot}(n_{maxCC}{+}1){\cdot}n$.*

4.4 Causal chain

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

Definition 11 (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 sequence of actions $a_1, a_2, \cdots, a_k$ 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 5 *Let GC be a connected component of G, v_0 be a process of GC, and seg be any GC-segment. Let $a_1, a_2, \cdots, a_k$ be a causal chain of seg rooted at v_0. Denote by v_i the process that executes a_i, for all $i \in \{1, \ldots, k\}$.*

- *For all $1 \leq i \leq k$, a_i consists in the execution of $computePath(v_i)$ (*i.e., *v_i executes the rule $\mathbf{R_{UN}}$ or $\mathbf{R_{RN}}$), where v_i is a process of GC.*
- *Assume a_1 is executed in the step $\gamma \mapsto \gamma'$ of seg. Denote by ds_1 the distance value of process v_0 in γ. For all $1 \leq i \leq k$, a_i sets d_{v_i} to $ds_1 \oplus w_{v_1}(v_0) \oplus \ldots \oplus w_{v_i}(v_{i-1})$.*

Lemma 11 *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 $\mathbf{R_{UN}}$ 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 $a_1, a_2, \cdots, a_k$ of a *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 $j > i$. The value of d_v is strictly larger in γ_{j+1} than in γ_i (Observation 5). This implies that process v must have executed either $\mathbf{R_{UR}}$, $\mathbf{R_I}$, $\mathbf{R_{RN}}$, or $\mathbf{R_{RR}}$ in the meantime.

If v has executed rule $\mathbf{R_{RR}}$ or $\mathbf{R_{UR}}$, then a_i and a_j are executed in two different segments by Lemmas 9 and 10, a contradiction.

Otherwise, v has status C in γ_i, so any following execution of the rule $\mathbf{R_{RN}}(v)$ or $\mathbf{R_I}(v)$ is preceded by the execution of the rule $\mathbf{R_{EF}}(v)$. Consequently, actions a_j and a_j executed in two different segments (Lemma 8), 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. □

4.4.1 Maximal Causal chains.

Definition 12 (Maximal 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 maximal causal chain of *seg rooted at v_0 is a causal chain $a_1, a_2, \cdots, a_k$ executed in seg such that the causal chain is maximal and the action a_1 sets $parent_{v_1}$ to v_0 not later than any action by v_0 in seg.*

Definition 13 ($SI_{seg,v}$) *Let GC be a connected component of G. Let v be a process of GC and a segment seg 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.

Observation 6 *Let GC be a connected component of G. Let v be a process of GC and a segment seg of GC. The size of the set $SI_{seg,v}$ is bounded by a function of the number of processes in GC.*

Following Observation 5, we have:

Observation 7 *Let GC be a connected component of G, $v_0 \in GC$, and seg be a GC-segment. Denote by ds_{seg,v_0} the distance value of v_0 at the beginning of seg.*

All actions of a maximal causal chain $a_1, a_2, \cdots, a_k$ of seg rooted at v_0 respectively sets d_{v_i} to $ds_{seg,v_0} \oplus w_{v_1}(v_0) \oplus \ldots \oplus w_i(v_{i-1})$, for all $i \in \{1, \ldots, k\}$.

Lemma 12 *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 rule $\mathbf{R_{UN}}$ executions done by u in seg is bounded by $X \cdot (n_{maxCC} - 1)$.*

Proof. First, assume that $\mathbf{R_{UN}}(u)$ is executed in some step $\gamma \mapsto \gamma'$ of seg and later in some other step $\gamma'' \mapsto \gamma'''$ of seg. By Corollary 3, any sequence of $\mathbf{R_{UN}}(u)$ executions in seg makes the value of d_u decreasing. Therefore, all the values of d_u obtained by the $\mathbf{R_{UN}}$ executions done by u are different. By definition of $SI_{seg,v}$ and Lemma 11, all these values belong to the set $\bigcup_{v \in GC \setminus \{u\}} SI_{seg,v}$, which has size at most $X \cdot (n_{\mathtt{maxCC}} - 1)$. □

By Theorem 4 and Lemma 12, we obtain the following result.

Corollary 4 *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 steps during any execution, is bounded by $(X \cdot (n_{maxCC} - 1) + 4) \cdot (n_{maxCC} + 1) \cdot n$.*

Let $\mathtt{W}_{\max} = \max\{w_u(v) : u \in V \wedge v \in \Gamma(u)\}$. If all weights are strictly positive integers and $\oplus$ is the addition operator, then the size of any $SI_{seg,u}$ is bounded by $\mathtt{W}_{\max}(n_{\mathtt{maxCC}} - 1)$ for all connected component GC, all GC-segment seg and all process $u \in GC$ because $S_{seg,u} \subseteq [ds_{seg,u} + 1, ds_{seg,u} + \mathtt{W}_{\max}(n_{cc} - 1)]$, where $n_{cc} \leq n_{\mathtt{maxCC}}$ is the number of processes in GC. Hence, we deduce the following theorem from Lemma 2, Corollary 1, and Corollary 4.

Theorem 5 *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 steps is at most $(\mathtt{W}_{\max} \cdot (n_{maxCC} - 1)^2 + 4) \cdot (n_{maxCC} + 1) \cdot n$.*

Lemma 13 *Let GC be a connected component of G, $v \in GC$, and seg be GC-segment. If all edges have the same weight, then $|SI_{seg,v}| < n_{maxCC}$.*

Proof. Assume that all edges have the same weight w. According to Observation 7, we have $SI_{seg,v} \subset \{ds_{seg,v} \oplus i.w \mid 1 \leq i \leq n_{\mathtt{maxCC}} - 1\}$. □

By Corollary 4 and Lemma 13, we obtain the following result.

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

5 Round Complexity of Scheme

5.1 From an Arbitrary Configuration to a Normal Configuration

Any process u that satisfies $P\_guardRR(u)$ (resp. $P\_updateRoot(u) \wedge st_u = C$) is enabled to execute rules either $\mathbf{R_{RR}}$ or $\mathbf{R_{RN}}$ (resp. either rules $\mathbf{R_{UN}}$ or $\mathbf{R_{UR}}$) and after executing one of these two rules, u satisfies both $\neg P\_guardRR(u)$ and $\neg P\_updateRoot(u) \vee st_u \neq C$. Hence, from Lemmas 3 and 5, we obtain the following result.

Lemma 14 *After at most 1 round, every process u satisfies $\neg P\_guardRR(u)$ and $\neg P\_updateRoot(u) \vee st_u \neq C$ forever, and consequently both rules $\mathbf{R_{UR}}$ and $\mathbf{R_{RR}}$ are disabled forever.*

To reach a normal configuration from an arbitrary configuration, Algorithm Scheme should also removed all illegal branches. The first lemma below essentially claims that all processes that are in illegal branches progressively switch to status EB within $n_{\texttt{maxCC}}$ rounds, in order of increasing depth.

Lemma 15 *Let $i \in \mathbb{N}^*$. From the beginning of round i, 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 ($i = 1$) is vacuum, so we assume that the lemma holds for some integer $i \geq 1$.

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

Then, let u be a process of status C at the beginning of round i. Each of its ancestors v is at depth smaller than i, have status EB, and have at least one child not having status EF. Thus, v cannot execute any rule, and consequently v 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.

Consider any process u with status C at depth i in an illegal branch at the beginning of the round i. 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, u will execute rule either $\mathbf{R_{EB}}$, $\mathbf{R_{UN}}$, $\mathbf{R_{UR}}$ 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 (*n.b.*, this latter case can only appear in round $i = 1$, see Lemma 14). This concludes the proof of the lemma. □

Corollary 6 *After at most n_{maxCC} 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 $n_{\texttt{maxCC}}$ rounds, in order of decreasing depth.

Lemma 16 *Let $i \in \mathbb{N}^*$. From the beginning of round $n_{maxCC} + i$, any process at depth larger than $n_{maxCC} - i + 1$ 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), so we assume that the lemma holds for some integer $i \geq 1$. At the beginning of round $n_{\texttt{maxCC}}+i$, any process at depth larger than $n_{\texttt{maxCC}} - i + 1$ has the status EF (by induction hypothesis). Therefore, processes with status EB at depth $n_{\texttt{maxCC}} - i + 1$ in an illegal branch can execute the rule $\mathbf{R_{EF}}$ at the beginning of round $n_{\texttt{maxCC}}+i$. These processes will thus all execute within round $n_{\texttt{maxCC}}+i$ (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 $n_{\texttt{maxCC}}$ has terminated, any process in an illegal branch that executes either gets status EF, or will be outside any illegal branch forever. □

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 17 *Let $i \in \mathbb{N}^*$. From the beginning of round $2n_{maxCC} + i$, there does not exist any process at depth larger than $n_{maxCC} - i + 1$ 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 $2n_{\texttt{maxCC}}+i$, no process is at depth larger than $n_{\texttt{maxCC}}-i+1$ in an illegal branch. All processes in an illegal branch have the status EF (by Lemma 16). So, at the beginning of round $2n_{\texttt{maxCC}} + i$, any abnormal root satisfies the predicate P_reset, they are enabled to execute either $\mathbf{R_I}$, or $\mathbf{R_{RN}}$. So, all abnormal roots at the beginning of the round $2n_{\texttt{maxCC}} + i$ 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 $2n_{\texttt{maxCC}} + i$. □

By Lemmas 14-17, we obtain the following result.

Theorem 6 *After at most $3n_{maxCC}$ rounds, a normal configuration of* Scheme *is reached.*

5.2 From a Normal Configuration to a Terminal Configuration

First, by Theorem 2, we have the following corollary.

Corollary 7 *In a normal configuration of* Scheme*, all processes in connected components containing no process satisfying canBeRoot are disabled.*

From the previous corollary, we only need to focus on any connected component GC containing at least one process satisfying $canBeRoot$. From a normal configuration, Algorithm Scheme needs additional rounds to propagate the status C and the correct distance in GC.

Definition 14 ($dmin_{GC}$) *Let $dmin_{GC} = \text{argmin}_{(u \in GC \wedge canBeRoot_u)}(distRoot(u))$.*

By definition, every process u satisfies $\neg P\_abnormalRoot(u)$ in a normal configuration. Hence we obtain the following observation.

Observation 8 *Let γ be a normal configuration of* Scheme*. Let u be a process such that $st_u = C$ in γ. Then, $P\_root(u)$ or $u \in Children(parent_u)$.*

Lemma 18 *Let γ be a normal configuration of* Scheme *and $u \in GC$. We have:*

1. *Only the rules $\mathbf{R_{UN}}(u)$ and $\mathbf{R_{RN}}(u)$ may be enabled in γ.*
2. *In γ, if $canBeRoot_u \wedge distRoot(u) = dmin_{GC}$ holds, then $P\_root(u)$ and $\neg P\_nodeImp(u)$ holds.*

Proof. By definition of γ, $st_u \notin \{EB, EF\}$ in γ, so $\mathbf{R_{EF}}(u)$ and $\mathbf{R_I}(u)$ are both disabled in γ. Moreover, $\neg P\_abnormalRoot(u) \wedge \forall v \in V, st_v \notin \{EB, EF\}$ implies that $\mathbf{R_{EB}}(u)$ is disabled in γ. Then, $\neg P\_updateRoot(u)$ implies that $\mathbf{R_{UR}}(u)$ is disabled in γ. Finally, as $canBeRoot_u \Rightarrow st_u \neq I$ in γ, we can deduce that $\mathbf{R_{RR}}(u)$ is disabled in γ. Hence, only $\mathbf{R_{UN}}(u)$ or $\mathbf{R_{RN}}(u)$ may be enabled in γ.

Assume, by contradiction, that $canBeRoot_u \wedge distRoot(u) = dmin_{GC}$, but $\neg P\_root(u)$ holds in γ. Then, by Observation 8, $parent_u \in \Gamma(u)$ and $u \in Children(parent_u)$ and, consequently, in configuration γ process u belongs to a legal branch (*n.b.*, $\forall x \in V, \neg P\_abnormalRoot(x)$), *i.e.*, a branch

rooted at a process v such that $P\_root(v)$. By definition, $d_u \succeq d_{parent_u} \oplus \omega_u(parent_u) \succ d_v = distRoot(v) \succeq dmin_{GC}$. Now, by definition of normal configuration, $d_u \preceq distRoot(u) = dmin_{GC}$. Hence, we obtain a contradiction: $P\_root(u)$ holds in γ. Finally, $P\_root(u)$ and $distRoot(u) = dmin_{GC}$ imply $\neg P\_updateNode(u)$, which in turn implies $\neg P\_nodeImp(u)$. □

Lemma 19 *Let γ be a normal configuration of* Scheme. *Let u be a process such that $st_u = C$ in γ. We have $d_u \succeq dmin_{GC}$. Moreover, any process v of GC satisfying $canBeRoot_v \wedge distRoot(v) = dmin_{GC}$ has reached its terminal state.*

Proof. First, by Observation 8, $P\_root(u) \vee u \in Children(parent_u)$ and, consequently, in configuration γ, process u belongs to a legal branch (*n.b.*, $\forall x \in V, \neg P\_abnormalRoot(x)$), *i.e.*, a branch rooted at a process w such that $P\_root(w)$ (maybe $u = w$). By definition, in γ, $d_u \succeq d_w = distRoot(w) \succeq dmin_{GC}$.

As $canBeRoot_v$ implies $st_v = C$ in γ (by definition of γ), and $canBeRoot_v \wedge distRoot(v) = dmin_{GC}$ implies $\neg P\_nodeImp(v)$ (Lemma 18.2), we can deduce, by Lemma 18.1, that v is disabled in γ, and by Theorem 1, we are done. □

Let $\mathtt{ST}_{GC}(i, \gamma) = \{u \mid$ u has its terminal state in the first configuration of the ith round from $\gamma\}$.

Lemma 20 *Let γ be a normal configuration of* Scheme *and $i \geq 0$. If $|V/\mathtt{ST}_{GC}(i, \gamma)| > 0$, then $|V/\mathtt{ST}_{GC}(i, \gamma)| > |V/\mathtt{ST}_{GC}(i + 1, \gamma)|$.*

Proof. Notice that $\mathtt{ST}_{GC}(i, \gamma)$ is not empty (Lemma 19). Assume that $|V/\mathtt{ST}_{GC}(i, \gamma)| > 0$. Let (u, v) be tuple such that $u \in \mathtt{ST}_{GC}(i, \gamma)$, $v \notin \mathtt{ST}_{GC}(i, \gamma)$, $u \in N(v)$ and (u, v) minimizes $d_u \oplus w_v(u)$ (the tuple (u, v) exists since $\mathtt{ST}_{GC}(i, \gamma)$ is not empty and GC is a connected component). After executing either $\mathbf{R_{UN}}(v)$ or $\mathbf{R_{RN}}(v)$, during the ith round, v reaches its terminal state, where $st_v = C \wedge d_v = d_u \oplus w_v(u)$ (Lemma 18). Hence, we can conclude that $|V/\mathtt{ST}_{GC}(i, \gamma)| > |V/\mathtt{ST}_{GC}(i+1, \gamma)|$. □

By Lemmas 2, 18-20, and Corollary 1, we have:

Corollary 8 *A terminal legitimate configuration of* Scheme *is reached in at most $n_{maxCC} - 1$ rounds from a normal configuration.*

By Theorem 6 and Corollary 8, we obtain the following result.

Corollary 9 *A terminal legitimate configuration of* Scheme *is reached in at most $4n_{maxCC} - 1$ rounds from any configuration.*

6 Instantiation

In this section, we illustrate the versatility of Algorithm Scheme by proposing several instantiations that solve various classical problems. Following the general bound (Corollary 9, page 21), all these instances reach a terminal configuration in at most $4n_{\mathtt{maxCC}} - 1$ 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).

Correctness of Forest. By Theorem 5 (page 18), Algorithm Forest self-stabilizes to a terminal legitimate configuration that satisfies the following requirements (see Definition 7, page 12).

Observation 9 *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 $u \in rootSet$,*
2. *there is a process of $rootSet$ in V_u, $st_u = C$, $parent_u \in \Gamma(u)$, and $d_u \geq d_{parent_u} + 1$,* 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.*

Step Complexity of Forest. Since for every process u, $P\_nodeImp(u) \equiv false$, rule $\mathbf{R_{UN}}$ is never enabled. Hence, the total number of steps during any execution is bounded by $4{\cdot}(n_{\texttt{maxCC}}{+}1){\cdot}n$, by Theorem 4 (page 16).

Algorithm 2: Parameters for any process u in Algorithm Forest

---

Inputs

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

Ordered Semigroup

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

Predicate

- $P\_nodeImp(u) \equiv false$

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.

Correctness of LEM. As *canBeRoot* is true for all processes, we can deduce, from Theorem 5 (page 18) and Definition 7 (page 12), that Algorithm LEM self-stabilizes to a terminal legitimate configuration that satisfies the following requirements.

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

1. $P\_root(u)$*, or*
2. $st_u = C$*,* $parent_u \in \Gamma(u)$*,* $d_u \succ d_{parent_u}$*.*

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 $st_u = st_v = C$, $\mathbf{R_{UN}}(u)$ is enabled, a contradiction. Hence, all processes of GC agree on the same leader identifier, and by definition, 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 10, $parent_u \in \Gamma(u)$ 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 10, 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 $st_u = C$, $\mathbf{R_{UN}}(u)$ or $\mathbf{R_{UR}}(u)$ is enabled, a contradiction.

Step Complexity of LEM. All edges have the same weight, so the total number of steps during any execution is bounded by $((n_{\texttt{maxCC}} - 1)^2 + 4) \cdot (n_{\texttt{maxCC}} + 1) \cdot n$ (Corollary 5, page 18), *i.e.*, $O({n_{\texttt{maxCC}}}^3 \cdot n)$.

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 in the connected component of G containing r. Moreover, in any other component, the nodes eventually detect the absence of r by taking the status I (Isolated).

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 Semigroup

- $DistSet = IDs \times \mathbb{N}$; 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

- $P\_nodeImp(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C \wedge d_v.id < d_u.id)$
- $P\_leader(u) \equiv P\_root(u)$

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 node u, the unique path from u to r in T is a shortest path from u to r in V_r.

Correctness of RSP.

Lemma 21 *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 12) and Corollary 1 (page 15), $st_r = C$ and $d_u \geq d_{parent_u} + 1 > 0$ in γ. So, $st_r = C \wedge P\_updateRoot(u)$ holds in γ, *i.e.*, either $\mathbf{R_{UN}}(r)$ or $\mathbf{R_{UR}}(r)$ is enabled in γ, a contradiction. □

By Lemma 21, Definition 7, and the fact that $\neg P\_nodeImp(u)$ holds for all process u such that $st_u = C$ (otherwise $\mathbf{R_{UN}}(u)$ is enabled), we obtain the following result.

Observation 11 *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 $st_u = I$.*

Step Complexity of RSP. All edges have a positive integer weight, so the total number of steps during any execution is bounded by $(\mathtt{W}_{\max} \cdot n_{\mathtt{maxCC}} \cdot (n_{\mathtt{maxCC}} - 1) + 4) \cdot (n_{\mathtt{maxCC}} + 1) \cdot n$ (Theorem 5, page 18), *i.e.*, $O({n_{\mathtt{maxCC}}}^3 \cdot n \cdot \mathtt{W}_{\max})$.

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 Semigroup

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

Predicate

- $P\_nodeImp(u) \equiv P\_updateNode(u)$

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

Assume the existence of a unique root r and that each process u distinguishes each of its neighbors v using a positive integer in $\{1 \dots \delta_u\}$ (*i.e.*, $\Gamma(u) = \{1 \dots \delta_u\}$, for all $u \in V$, and $Lbl = \{1, \dots, \Delta\}$, where Δ is the maximum degree in the graph). Algorithm RDFS is the instantiation of Scheme with the parameters given in Algorithm 5. Algorithm RDFS computes (in a self-stabilizing manner) a depth-first search (DFS) tree spanning in the connected component of G containing r. Moreover, in any other component, nodes 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 $d^r_{lex}(u)$, is the minimum weight of a path from u to r (according to the lexicographical order).

Correctness of RDFS.

Observation 12 *Let T be a tree rooted at r that spans V_r. Following the result of [43], if for every process $u \in V_r$, the weight of the path from u to r in T is equal to $d^r_{lex}(u)$, 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 13 *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 $st_u = I$.*

Step Complexity of RDFS. Let GC be a connected component of G. Let seg be GC-segment and v a node of GC. According to Observation 7 (page 17), we have $|SI_{seg,v}| \leq \Delta \cdot n_{\texttt{maxCC}}$. Hence, the total number of steps during any execution is bounded by $(\Delta \cdot n_{\texttt{maxCC}} \cdot (n_{\texttt{maxCC}} - 1) + 4) \cdot (n_{\texttt{maxCC}} + 1) \cdot n$ (Corollary 4, page 18), *i.e.*, $O(\Delta \cdot {n_{\texttt{maxCC}}}^3 \cdot n)$.

Algorithm 5: 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 Semigroup

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

Predicate

- $P\_nodeImp(u) \equiv P\_updateNode(u)$

6.5 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 6. 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.

Correctness of LEM_BFS. As *canBeRoot* is true for all processes, we can deduce, from Theorem 5 (page 18) and Definition 7 (page 12), that in a terminal configuration $st_u = C$, for every process u. Then, following the same reasoning as for Algorithm LEM and owing the fact that $\neg P\_nodeImp(u)$ holds for all process u in a terminal configuration (otherwise $\mathbf{R_{UN}}(u)$ is enabled), follows.

Observation 14 *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.

Step Complexity of LEM_BFS**.** All edges have the same weight, so the total number of steps during any execution is bounded by $((n_{\texttt{maxCC}} - 1)^2 + 4) \cdot (n_{\texttt{maxCC}} + 1) \cdot n$ (Corollary 5, page 18), *i.e.*, $O({n_{\texttt{maxCC}}}^3 \cdot n)$.

Algorithm 6: Parameters for any process u in Algorithm LEM_BFS

Inputs

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

Ordered Semigroup

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

Predicates

- $P\_nodeImp(u) \equiv P\_updateNode(u)$
- $P\_leader(u) \equiv P\_root(u)$

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 $4n_{\texttt{maxCC}} - 1$ rounds, where $n_{\texttt{maxCC}}$ is the maximum number of processes in a connected component. We exhibited polynomial upper bounds on its stabilization time in steps that depend on the instantiation we consider. We illustrated the versatility of our approach by proposing several instantiations of Scheme that solve various classical problems. 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 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, or DFS trees. Assuming now an input set of roots, we also proposed an instance to compute a spanning forest of arbitrary shaped trees, with non-rooted components detection. Again, one can easily enforce this latter construction to obtain BFS, DFS, or shortest-path forests. Finally, assuming a rooted network, we proposed shortest-path and DFS spanning tree constructions, 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 Technolog (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] 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.

[23] 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.

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

[25] Paolo Boldi and Sebastiano Vigna. Universal dynamic synchronous self–stabilization. *Distributed Computing*, 15(3):137–153, July 2002.

[26] 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.

[27] 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.

[28] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, 2001.

[29] 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.

[30] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

[31] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *TAAS*, 4(1):6:1–6:27, 2009.

[32] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), 2009.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] M Sloman and J Kramer. *Distributed systems and computer networks.* Prentice Hall, 1987.

[39] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only Read/Write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[40] 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.

[41] Ernest J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.

[42] Adrian Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.

[43] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Inf. Process. Lett.*, 49(6):297–301, 1994.