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
Univ. Grenoble Alpes, CNRS,
Grenoble INP,† VERIMAG
Grenoble, France
stephane.devismes@
univ-grenoble-alpes.fr

David Ilcinkas
CNRS & Univ. Bordeaux,
LaBRI, UMR 5800
Talence, France
ilcinkas@labri.fr

Colette Johnen
Univ. Bordeaux,
LaBRI, CNRS UMR 5800
Talence, France
johnen@labri.fr

ABSTRACT

In this paper, we propose a general scheme, called Algorithm STlC, to compute spanning-tree-like data structures on arbitrary networks. STlC 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 $O(n_{\mathsf{maxCC}})$ rounds, where n_{maxCC} is the maximum number of processes in a connected component. We also exhibit polynomial upper bounds on its stabilization time in steps and process moves holding for large classes of instantiations of Algorithm STlC.

We illustrate the versatility of our approach by proposing several such instantiations that efficiently solve classical problems such as leader election, as well as, unconstrained and shortest-path spanning tree constructions.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**;

KEYWORDS

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

ACM Reference Format:
Stéphane Devismes, David Ilcinkas, and Colette Johnen. 2018. Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions. In *Proceedings of International Conference on Distributed Computing and Networking (ICDCN'19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A *self-stabilizing algorithm* 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 (such faults may corrupt the configuration of the system), provided that those faults do not alter the code of the processes. Among the vast self-stabilizing literature, many works (see [29] 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 actually achieve an additional property called *silence* [27]: a silent self-stabilizing algorithm converges within finite time to a configuration from which the values of the communication variables used by the algorithm remain fixed. Silence is a desirable property. Indeed, as noted in [27], 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 [37] since it allows to simplify both the design and the proofs of self-stabilizing algorithms. Various composition techniques have been introduced so far, *e.g.*, collateral composition [32], fair composition [25], cross-over composition [4], and conditional composition [18]; and many self- stabilizing algorithms are actually made as a composition of a silent spanning-tree-like construction and another algorithm designed for tree/forest topologies, *e.g.*, [3, 7, 17]. Notably, the silence property is not mandatory in such designs, however it allows to write simpler proofs. Finally, notice that silent spanning-tree-like constructions have also been used to build very general results, *e.g.*, the self-stabilizing proof-labeling scheme constructions proposed in [6].

We consider the locally shared memory model with composite atomicity introduced by Dijkstra [24], which is the most commonly used model in self-stabilization. In this model, executions proceed in atomic steps (where a subset of enabled processes move, *i.e.*, update their local states) and the asynchrony of the system is captured by the notion of *daemon*. The weakest (*i.e.*, the most general) daemon is the *distributed unfair daemon*. Hence, solutions stabilizing under such an assumption are highly desirable, because they work under any other daemon assumption. Moreover, the *stabilization time* can also be bounded in terms of steps (and moves) only 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 speeds of the different processes are roughly equal, then the execution time is

*This study has been partially supported by the ANR projects DESCARTES (ANR-16-CE40-0023) and ESTATE (ANR-16-CE25-0009).

† Institute of Engineering Univ. Grenoble Alpes.

ICDCN'19, January 2019, Bangalore, India

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of the order of magnitude of the round complexity. Otherwise, if the system is truly asynchronous, then the execution time is of the order of magnitude of the step complexity. The stabilization time in moves captures the amount of computations an algorithm needs to recover a correct behavior. Notice that the number of moves and the number of steps are closely related: if an execution e contains x steps, then the number y of moves in e satisfies $x \leq y \leq n \cdot x$, where n is the number of processes. [1] Finally, if, for example, an algorithm is self-stabilizing under a weakly fair daemon, but not under an unfair one, then this means that the stabilization time in moves cannot be bounded, so there are processes whose moves do not make the system progress in the convergence. In other words, these processes waste computation power and so energy. Such a situation should therefore be prevented, making algorithms withstanding the unfair daemon more desirable than the weakly fair one.

There are many self-stabilizing algorithms proven under the distributed unfair daemon, *e.g.*, [2, 8, 19, 20, 30]. However, analyses of the stabilization time in steps (or moves) is rather unusual and this may be an important issue. Indeed, recently, several self-stabilizing algorithms which work under a distributed unfair daemon have been shown to have an exponential stabilization time in steps in the worst case. In [2], silent leader election algorithms from [19, 20] are shown to be exponential in steps in the worst case. In [23], the Breadth-First Search (BFS) algorithm of Huang and Chen [33] is also shown to be exponential in steps. Finally, in [31] authors show that the silent self-stabilizing algorithm they proposed in [30] is also exponential in steps.

1.1 Contribution

We propose a general scheme, called Algorithm STIC (stands for Spanning-Tree-like Constructions), to compute spanning-tree-like data structures on bidirectional weighted networks of arbitrary topology. Algorithm STIC 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 STIC is efficient. Indeed, its stabilization time is at most $4n_{\mathtt{maxCC}}$ rounds, where $n_{\mathtt{maxCC}}$ is the maximum number of processes in a connected component. Moreover, its stabilization time in moves is polynomial in the usual cases (see the example instantiations we propose). Precisely, we exhibit polynomial upper bounds on its stabilization time in moves that depend on the particular problems we consider. To illustrate the versatility of our approach, we present here four instantiations of STIC solving classical spanning-tree-like problems.

- Assuming an input set of roots, we propose an instance to compute a spanning forest of arbitrary shaped trees, with non-rooted components detection.
  By non-rooted components detection, we mean that every process that belongs to a connected component which does not contain a root should eventually take a special state notifying that it detects the absence of a root.
  This instance stabilizes in $O(n_{\mathtt{maxCC}} \cdot n)$ moves, which matches the best known step complexity for spanning tree construction [12] with explicit parent pointers.
  Actually, there exists a solution with implicit parent pointer [34] that achieves a better complexity, $O(n \cdot D)$ moves, where D is the network diameter. However adding a parent pointer to this algorithm makes this solution more costly than ours in a large class of networks, as we will explain later.
- Assuming a rooted network, we propose a shortest-path spanning tree construction, with non-rooted components detection, that stabilizes in $O({n_{\mathtt{maxCC}}}^3 \cdot n \cdot \mathtt{W}_{\max})$ moves, where $\mathtt{W}_{\max}$ is the maximum weight of an edge.
  Again, this move complexity matches the best known move complexity for this problem [21].
- Finally, assuming processes have unique identifiers, we propose two instantiations of STIC, 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. The move complexity of this version is in $O({n_{\mathtt{maxCC}}}^2 \cdot n)$ moves. This bound matches the best known step complexity for leader election [2].
  - In the second version, the trees are BFS. This latter version stabilizes in $O({n_{\mathtt{maxCC}}}^3 \cdot n)$ moves.

From these various examples, one can easily derive other silent self-stabilizing spanning-tree-like constructions, *e.g.*, silent Depth-First Search (DFS) spanning tree constructions.

1.2 Related Work

The locally shared memory model with composite atomicity is the standard model in self-stabilization [24]. Another (more realistic) model is the read/write atomicity [28], where an atomic step of a process consists of an internal computation followed by either a read or write action on shared registers, but not both. Dolev *et al.* [28] have presented a method to transform any algorithm which is self-stabilizing under composite atomicity into an algorithm that stabilizes to the same specification in the presence of read/write atomicity. Then, Afek and Brown in [1] have presented a self-stabilizing version of Alternating bit protocol (ABP), while Dolev *et al.* [26] proposes a stabilizing data-link protocol that emulates a reliable FIFO communication channel over unreliable capacity bounded non-FIFO channels. So by combining the work of [28] with the work of [1] or [26], one can transform a self-stabilizing algorithm designed in shared memory model with composite atomicity into a protocol that is stabilizing for the same specification yet in message passing model with bidirectional communication, see [25] for details.

Several works consider the design of particular spanning-tree-like constructions in the locally shared memory model with composite atomicity and their move/step complexity.

Self-stabilizing algorithms that construct BFS trees in arbitrary connected and rooted networks are proposed in [15, 16]. The algorithm in [15] is not silent and has a stabilization time of $O(\Delta \cdot n^3)$ steps (Δ is the maximum degree of the network). The silent algorithm given in [16] has a stabilization time of $O(D^2)$ rounds and $O(n^6)$ steps.

Silent self-stabilizing algorithms that construct spanning trees of arbitrary topologies in arbitrary connected and rooted networks are given in [12, 34]. The solution proposed in [12] stabilizes in

[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.

at most $4 \cdot n$ rounds and $5 \cdot n^2$ steps, while the algorithm given in [34] stabilizes in $n \cdot D$ moves. However, its round complexity is not analyzed and the parent of a process is not computed explicitly. Now, Cournier [11] showed that the straightforward variant of this algorithm where a parent pointer variable is added has a stabilization time of $\Omega(n^2 \cdot D)$ steps in an infinite class of networks.

Several papers propose self-stabilizing algorithms stabilizing in both a polynomial number of rounds and a polynomial number of steps, *e.g.*, [2] (for the leader election in arbitrary connected networks of processes having unique identifiers), and [13, 14] (for the DFS token circulation in arbitrary connected and rooted networks). The silent leader election algorithm proposed in [2] stabilizes in at most $3 \cdot n + D$ rounds and $O(n^3)$ steps. DFS token circulations given in [13, 14] 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 [13], processes are additionally assumed to have unique identifiers.

In [9], a generic self-stabilizing algorithm is presented for constructing in a rooted connected network a spanning tree where a given metric is maximized. Now, since the network is assumed to be rooted (*i.e.*, a leader node is already known), leader election is not an instance of their generic algorithm. Similarly, since they assume connected networks, the non-rooted components detection cannot be expressed too. Finally, the algorithm is proven assuming the restricted centralized weakly-fair daemon. In particular, step complexity cannot bounded under such an assumption.

In [21], we presented a self-stabilizing shortest-path tree construction with a single root in composite atomicity model with distributed unfair daemon; the algorithm is efficient both in terms of rounds and moves, and tolerates disconnections. We generalize this approach here with the goal of keeping similar efficient complexity bounds. It is worth noting that this purpose was challenging since general schemes and efficiency are usually understood as orthogonal issues.

1.3 Roadmap

The remainder of the paper is organized as follows. In the next section, we present the computational model and basic definitions. In Section 3, we describe our algorithm STlC and justify its round complexity. Its proof of correctness and a complexity analysis in moves are sketched in Section 4 (due to the lack of space, the complete proofs have been omitted, they are available in the technical report online [22]). Several instantiations of STlC with their specific complexity analyses are presented in Section 5. Finally, we make concluding remarks in Section 6.

2 PRELIMINARIES

2.1 Distributed Systems

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 in 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 [36]. 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 $v \in \Gamma(u)$).

2.2 Computational Model

We use the *composite atomicity model of computation* in which the processes communicate using a finite number of locally shared variables, henceforth simply 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

$$\langle label \rangle \ : \ \langle guard \rangle \ \rightarrow \ \langle action \rangle$$

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,[2] leading to a new configuration γ', and so on. The transition from γ to γ' is a *step* where processes of $\mathcal{X}$ execute a *move*. The possible steps induce a binary relation over the set of configurations, denoted by $\mapsto$. An *execution* is a maximal sequence of configurations $e = \gamma_0 \gamma_1 \cdots \gamma_i \cdots$ 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.

[2] In case of several enabled actions at the activated process, the choice of the executed action is nondeterministic.

2.3 Silent Self-Stabilization

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

(1) all executions under S are finite, and
(2) all terminal configurations belong to $\mathcal{L}$.

2.4 Time Complexity

We measure the time complexity of an algorithm using two notions: *rounds* [28] and *moves* [24].

We say that a process *moves* in $\gamma_i \mapsto \gamma_{i+1}$ when it executes a rule in $\gamma_i \mapsto \gamma_{i+1}$.

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 moves, steps, or rounds) over every execution possible under the considered daemon S (starting from any initial configuration) to reach a terminal (legitimate) configuration.

3 ALGORITHM STIC AND ITS ROUND COMPLEXITY

3.1 The problem

We propose a general silent self-stabilizing algorithm, called STIC (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 input constants.

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

Then, according to the specific problem we consider, we may want to minimize the weight of the trees using some kind of distance. To that goal, we assume that each edge $\{u, v\}$ has two *weights*: $\omega_u(v)$ denotes the weight of the arc (u, v) and $\omega_v(u)$ denotes the weight of the arc (v, u). Both values belong to the domain $DistSet$.

Let $(DistSet, \oplus, \prec)$ be an ordered magma, *i.e.*, $\oplus$ is a closed binary operation on $DistSet$ and $\prec$ is a total order on this set. The definition of $(DistSet, \oplus, \prec)$ is problem dependent and, if necessary, *i.e.*, if the problem dependent predicate $P\_nodeImp(.)$ holds ($P\_nodeImp(v)$ is true if process v is required to act to minimize the weight of the tree), the weight of the trees will be minimized using the ordered magma and the distance values that each candidate u should take when it is the root of a tree. This latter value is given by the (problem dependent) function $distRoot(u)$. Finally, 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 $d \prec d \oplus \omega_v(u)$.

Notice that the ordered magma is necessary for the versatility of our solution as it allows STIC to compute various kinds of spanning tree constructions. For example, following the approach proposed in [10], we can instantiate the ordered magma with the set of strings as domain, the concatenation as operator, and the lexicographical order. Then, STIC can compute as distance value the sequence of the channel port numbers traversed by a path. Consequently, a spanning tree where these distances are minimized according to the lexicographic order (using $P\_nodeImp(.)$) will be built, *i.e.*, a depth-first spanning tree (see algorithm RDFS in the technical report online [22]).

3.2 The variables

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

- $st_u \in \{I, C, EB, EF\}$**:** this variable gives the *status* of the process. I, C, EB, and EF respectively stand for *Isolated*, *Correct*, *Error Broadcast*, and *Error Feedback*.
  The two first status, I and C, are involved in the normal behavior of the algorithm, while the two last ones, EB and EF, are used during the correction mechanism. The meaning of EB and EF will be further detailed in Subsection 3.4.
  In a terminal configuration, if V_u contains a candidate, then $st_u = C$, otherwise $st_u = I$.
- $parent_u \in \{\perp\} \cup Lbl$**:** In a terminal configuration, if V_u contains a candidate, then either $parent_u = \perp$, *i.e.*, u is a tree root, or $parent_u$ belongs to $\Gamma(u)$, *i.e.*, $parent_u$ designates a neighbor of u, referred to as its *parent*. Otherwise (V_u does not contain a candidate), the value of $parent_u$ is meaningless.
- $d_u \in DistSet$**:** In a terminal configuration, if V_u contains a candidate, then d_u is larger than or equal to the weight of the tree path from u to its tree root, otherwise the value of d_u is meaningless.

Using these variables, we define the legitimate configurations of STIC as follows.

Definition 3.1 (Legitimate state and configuration). A *legitimate configuration* of STIC is a configuration where every process is in

Algorithm 1: Algorithm STlC, 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:

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

Macros:

- $Children(u) = \{v \in \Gamma(u) \mid st_v \neq I \wedge parent_v = u \wedge d_v \geq d_u \oplus \omega_v(u) \wedge (st_v = st_u \vee st_u = EB)\}$
- $beRoot(u)$: $st_u := C$; $parent_u := \perp$; $d_u := distRoot(u)$;
- $computePath(u)$:
  if $\{v \in \Gamma(u) \mid st_v = C\} \neq \emptyset$ **then**
  $st_u := C$;
  $parent_u := \text{argmin}_{(v \in \Gamma(u) \mid st_v = C)}(d_v \oplus \omega_u(v))$;
  $d_u := d_{parent_u} \oplus \omega_u(parent_u)$;
  if $P\_updateRoot(u)$ **then** $beRoot(u)$;
  else
  $beRoot(u)$;
  end if

Rules:

$\mathbf{R_U}(u)$:	$st_u = C \wedge P\_nodeImp(u)$	$\rightarrow$	$computePath(u)$;
$\mathbf{R_{EB}}(u)$:	$st_u = C \wedge \neg P\_nodeImp(u) \wedge (P\_abnormalRoot(u) \vee st_{parent_u} = EB)$	$\rightarrow$	$st_u := EB$;
$\mathbf{R_{EF}}(u)$:	$st_u = EB \wedge (\forall v \in Children(u) \mid st_v = EF)$	$\rightarrow$	$st_u := EF$;
$\mathbf{R_I}(u)$:	$P\_reset(u) \wedge \neg canBeRoot_u \wedge (\forall v \in \Gamma(u) \mid st_v \neq C)$	$\rightarrow$	$st_u := I$;
$\mathbf{R_R}(u)$:	$(P\_reset(u) \vee st_u = I) \wedge [canBeRoot_u \vee (\exists v \in \Gamma(u) \mid st_v = C)]$	$\rightarrow$	$computePath(u)$;

a legitimate state. A process u is said to be in a *legitimate state* of STlC 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, $st_u = C$, $parent_u \in \Gamma(u)$, $d_u \geq d_{parent_u} \oplus \omega_u(parent_u)$, and $\neg P\_nodeImp(u)$, or
(3) there is no process satisfying *canBeRoot* in V_u and $st_u = I$.

Let u be a process. In a legitimate configuration, either no process satisfies *canBeRoot* in V_u and every process v in V_u has status $st_v = I$; or there is at least one candidate and every process in V satisfies one of the two first conditions in Definition 3.1. Moreover, in this latter case, at least one process v in V_u satisfies $P\_root(u)$, that is, $canBeRoot_v$, $st_v = C$, $parent_v = \perp$, and $d_v = distRoot(v)$. In other word, v is a tree root.

One can establish that any terminal configuration is legitimate (and that a legitimate configuration is terminal, see Lemma 1 in the technical report online [22]).

Theorem 3.2. *Any terminal configuration of* STlC *is legitimate.*

3.3 Typical Execution

Assume the system starts from a configuration where, for every process u, $st_u = I$. All processes that belong to a connected component containing no candidates are disabled forever. Focus now on a connected component GC where at least one process is a candidate.

Then, any process u of status I that is a candidate or a neighbor of a process of status C is enabled to execute rule $\mathbf{R_R}$: it eventually executes $\mathbf{R_R}(u)$ to initiate a tree or to join a tree rooted at some candidate, choosing among the different possibilities the one that minimizes its distance value. Using this rule, it also switches its status to C and sets d_u to either $distRoot(u)$, or $d_v \oplus \omega_u(v)$ if it chooses a parent v. Executions of rule $\mathbf{R_R}$ are asynchronously propagated in GC until all processes of GC have status C. In parallel, rules $\mathbf{R_U}$ are executed to reduce the weight of the trees, if necessary: when a process u with status C satisfies $P\_nodeImp(u)$, this means that u can reduce d_u by selecting another neighbor with status C as parent and this reduction is required by the specification of the problem to be solved ($P\_nodeImp(u)$ is problem dependent). In this case, u chooses the neighbor which allows to minimize the value of d_u. In particular, notice that a candidate can lose its tree root condition using this rule, if it finds a sufficiently good parent in its neighborhood. Overall, within at most $n_{\mathtt{maxCC}}$ rounds, a terminal configuration is reached in which a specific spanning forest (maybe a spanning tree) is defined in each connected component containing at least one candidate, while in other components all processes are isolated.

3.4 Error Correction

Assume now that the system is in an arbitrary configuration. Inconsistencies between the states of the processes are detected using predicate $P\_abnormalRoot$.

Definition 3.3 (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*.

Informally, a process u is an *abnormal root* if u is neither a normal root (*i.e.*, $\neg P\_root(u)$), nor isolated (*i.e.* $st_u \neq I$), and satisfies one of the following three conditions:

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

Every process u that is neither an abnormal root nor isolated satisfies one of the two following cases.

- Either u is a normal root, *i.e.*, $P\_root(u)$,
- or u points to some neighbor (*i.e.*, $parent_u \in \Gamma(u)$) and the state of u is coherent *w.r.t.* the state of its parent.
  In this latter case, $u \in Children(parent_u)$, *i.e.*, u is a "real" child of its parent. See the definition below.

Definition 3.4 (Children). For every process v, $Children(v) = \{u \in \Gamma(v) \mid st_u \neq I \wedge parent_u = v \wedge d_u \geq d_v \oplus \omega_u(v) \wedge (st_u = st_v \vee st_v = EB)\}$.

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 $Children(v)$, where $v = parent_u$).*

Definition 3.5 (Branch). Consider a path $\mathcal{P} = u_0, \ldots, u_k$ such that $\forall i, 0 \leq i < k$, $u_{i+1} \in Children(u_i)$. $\mathcal{P}$ is acyclic. If u_0 is either a normal or an abnormal root, then $\mathcal{P}$ is called a *branch* rooted at u_0. The process u_i is said to be at *depth* i and $u_i, \ldots, u_k$ is called a *sub-branch*. If u_0 is an abnormal root, the branch is said to be *illegal*, otherwise, the branch is said to be *legal*.

Observation 2.

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

Definition 3.6 (Tree). 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*.

Definition 3.7 (Normal Configuration). We call any configuration without abnormal trees a *normal configuration*.

So, to recover a normal configuration, it is necessary to remove all abnormal trees. For each abnormal tree T, we have two cases. If the abnormal root u of T can join another tree T' using rule $\mathbf{R_U}(u)$ (thus without increasing its distance value, since, in this case, $P\_nodeImp(u)$ holds), then it does so and T disappears by becoming a subtree of T'. Otherwise, T is entirely removed in a top-down manner, starting from its abnormal root u. Now, in that case, we have to prevent the following situation: u leaves T; this removal creates some abnormal trees, each of those being rooted at a previous child of u; and later u joins one of those (created) trees or a tree issued from them. (This issue is sometimes referred to as the count-to-infinity problem [35].) Hence, the idea is to freeze T, before removing it. By freezing we mean assigning each member of the tree to a particular state, here EF, so that

(1) no member v of the tree is allowed to execute $\mathbf{R_U}(v)$, and
(2) no process w can join the tree by executing $\mathbf{R_R}(w)$ or $\mathbf{R_U}(w)$.

Once frozen, the tree can be safely deleted from its root to its leaves.

The freezing mechanism (inspired from [5]) is achieved using the status EB and EF, and the rules $\mathbf{R_{EB}}$ and $\mathbf{R_{EF}}$. If a process is not involved into any freezing operation, then its status is I or C. Otherwise, it has status EB or EF and no neighbor can select it as its parent. These two latter status are actually used to perform a "Propagation of Information with Feedback" 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 $\mathbf{R_{EB}}$. The propagation of the status EB in a top-down manner is done in at most $n_{\mathtt{maxCC}}$ rounds (see Lemma 14 in the technical report online [22]). Then, once the EB wave reaches a leaf, the leaf initiates a convergecast EF-wave using rule $\mathbf{R_{EF}}$. The propagation of the status EF in a bottom-up manner is also done in at most $n_{\mathtt{maxCC}}$ rounds (see Lemma 15 in the technical report online [22]). Once the EF-wave reaches the abnormal root, the tree is said to be *dead*, meaning that all processes in the tree have status EF and, consequently, no other process can join it. So, the tree can

be safely deleted from its abnormal root toward its leaves. There are several possibilities for the deletion depending on whether or not the process u to be deleted is a candidate or has a neighbor with status C. If u is a candidate and has no neighbor with status C: u becomes a normal root by executing $beRoot(u)$ using the rule $\mathbf{R_R}(u)$. If u has a neighbor with status C, again the rule $\mathbf{R_R}(u)$ is executed: u tries to directly join another "alive" tree, however if u is candidate and becoming a normal root allows it to further minimize d_u, it executes $beRoot(u)$ to become a normal root. If u is not a candidate and has no neighbor with status C, the rule $\mathbf{R_I}(u)$ is executed: u becomes isolated, and might join another tree later.

The removal of all dead trees requires at most n_{maxCC} rounds (see Lemma 16 in the technical report online [22]). Hence, overall after at most $3n_{\mathsf{maxCC}}$ rounds, any execution has reached a *normal configuration*, meaning that no process has status EB or EF, and there are no abnormal roots.

Lemma 3.8. *After at most $3n_{\mathsf{maxCC}}$ rounds, a normal configuration of* STIC *is reached.*

Then, similarly to the typical execution presented in Subsection 3.3 and using the properties of *P_nodeImp*, we showed that n_{maxCC} additional rounds are necessary to reach a terminal configuration from a normal configuration (see Corollary 7 in the technical report online [22]), hence follows.

Corollary 3.9. *A terminal and legitimate configuration of any instantiation of* STIC *is reached in at most $4n_{\mathsf{maxCC}}$ rounds from any configuration.*

Let u be a process belonging to an abnormal tree of which it is not the root. Let v be its parent. From the previous explanation, it follows that during the correction, $(st_v, st_u) \in \{(C,C), (EB,C), (EB,EB), (EB,EF), (EF,EF)\}$ until v resets by $\mathbf{R_R}(v)$ or $\mathbf{R_I}(v)$. Now, due to the arbitrary initialization, the status of u and v may not be coherent, in this case u is an abnormal root. Precisely, as formally defined in Algorithm 1, the status of u is incoherent *w.r.t.* the status of its parent v if $st_u \neq st_v$ and $st_v \neq EB$. For example, if a process u belongs to a tree (*i.e.*, $st_u \neq I$) and designates an isolated node v with $parent_u$ (*i.e.*, $parent_u = v$ and $st_v = I$), then the status of u is incoherent *w.r.t.* its parent v, *i.e.*, u is actually an abnormal root.

Finally, notice that the freezing mechanism ensures that if a process is the root of an alive abnormal tree, it is in that situation since the initial configuration (see Lemma 5 in the technical report online [22]). As explained in the next section, the bounded move complexity mainly relies on this strong property.

4 MOVE AND STEP COMPLEXITIES

4.1 GC-segment

In the following, 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).

Let GC be a connected component of G. The notion of *GC-segment* defined below helps us to analyze the number of moves per process.

Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in the configuration γ, u is an alive abnormal root, or $P\_updateRoot(u) \wedge st_u = C$ holds. We first prove (see Lemmas 5 and 6 in the technical report online [22]) that if a process satisfies one of these two conditions, then it does so from the beginning of the execution. Let $e = \gamma_0\gamma_1\cdots$ be an execution of STIC. We thus have $SL(\gamma_{i+1}, GC) \subseteq SL(\gamma_i, GC)$, for all $i \geq 0$.

Definition 4.1 (Segments). The *first GC-segment* of $e = \gamma_0\gamma_1\cdots$ is the maximal prefix $\gamma_0\cdots\gamma_i\gamma_{i+1}$ of e, such that $SL(\gamma_i, GC) = SL(\gamma_0, GC)$. The *second GC-segment* of e is the first GC-segment of the suffix $\gamma_{i+1}\gamma_{i+2}\cdots$, and so forth.

Notice that along any execution, the number of GC-segments is bounded by $n_{\mathsf{maxCC}} + 1$.

Let u be any process of GC. We prove (see Corollary 2 in the technical report online [22]) that the sequence of rules executed by u during a GC-segment belongs to the following language:

$$(\mathbf{R_I} + \varepsilon)(\mathbf{R_R} + \varepsilon)(\mathbf{R_U})^*(\mathbf{R_{EB}} + \varepsilon)(\mathbf{R_{EF}} + \varepsilon)$$

Hence, we deduce the following result.

Theorem 4.2. *If the number of $\mathbf{R_U}$ executions during a GC-segment by any process of GC is bounded by nb_UN, then the total number of moves (and steps) in any execution is bounded by*

$$(nb\_UN + 4)(n_{\mathsf{maxCC}} + 1)n$$

4.2 Maximal Causal Chain

The notion of maximal causal chain helps us to analyze the number of $\mathbf{R_U}$ executions in a GC-segment seg.

Definition 4.3 (Maximal Causal Chain). A *maximal causal chain of seg rooted at v_0* is a non-empty maximal sequence of *computePath* actions $a_1, a_2, \ldots, a_k$ executed in seg such that

(1) the action a_1 sets $parent_{v_1}$ to v_0 not later than any $\mathbf{R_U}$ action by v_0 in seg and
(2) 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.

As an $\mathbf{R_U}$ action always decreases the distance value, all actions in a maximal causal chain of seg rooted at v_0 are caused by distinct processes, different from v_0 (see Lemma 8 in the technical report online [22]). So the length of any maximal causal chain is less than n_{maxCC}. Coupled with additional properties of our algorithm, this allows us to show (see Lemma 10 in the technical report online [22]) that the number of $\mathbf{R_U}$ executions during seg by any process of GC is bounded, in general, by $n_{\mathsf{maxCC}}!$ (the factorial of n_{maxCC}), leading to Theorem 4.4 below. However, note that this huge complexity can be refined to obtain polynomial complexities in many practical cases, see Theorem 4.5.

Theorem 4.4. *Algorithm* STIC *is silent self-stabilizing under the distributed unfair daemon and has a bounded move (and step) complexity.*

Assume that weights are strictly positive integers bounded by $\mathtt{W}_{\max}$ and $\oplus$ is the addition operator. The distance values set by an action in a maximal causal chain rooted at some process u belong to the interval $[ds_{seg,u} + 1, ds_{seg,u} + \mathtt{W}_{\max}(n_{\mathsf{maxCC}} - 1)]$, where $ds_{seg,u}$ is the initial value of u in seg. We can deduce from this observation the following result (for more details see Subsection 4.5 in the technical report online [22]).

Theorem 4.5. *When all weights are strictly positive integers bounded by* $\mathtt{W}_{\max}$*, and* $\oplus$ *is the addition operator, the stabilization time of* STIC *in moves (and steps) is at most*

$$(\mathtt{W}_{\max}(n_{\mathtt{maxCC}} - 1)^2 + 5)(n_{\mathtt{maxCC}} + 1)n$$

5 INSTANTIATIONS

We now illustrate the versatility of Algorithm STIC by proposing several instantiations that solve various classical problems. Following the general bound (Corollary 3.9), all these instances reach a terminal configuration in at most $4n_{\mathtt{maxCC}}$ rounds, starting from an arbitrary one. In the following, we also address the specific move complexity of each proposed instance.

Notice that many other instantiations can be envisioned (see the conclusion and the technical report online [22]).

5.1 Spanning Forest and Non-Rooted Components Detection

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

Algorithm 2: Parameters for any process u in Algorithm Forest

Inputs:

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

Ordered Magma:

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

Predicate:

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

5.1.1 Correctness of Forest. By Theorems 3.2 (page 5) and 4.4 (page 7), Algorithm Forest self-stabilizes to a terminal legitimate configuration that satisfies the following requirements (see Definition 3.1).

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

5.1.2 Move Complexity of Forest. Since for every process u, $P\_nodeImp(u) \equiv P\_updateRoot(u)$, rule $\mathbf{R_U}$ is enabled at most once. Hence, the total number of moves (and steps) during any execution is bounded by $5(n_{\mathtt{maxCC}} + 1)n$, by Theorem 4.2 (page 7).

5.2 Leader Election

Assuming the processes have unique identifiers, Algorithm LEM is the instantiation of STIC with the parameters given in Algorithm 3. In each connected component, Algorithm LEM elects the process u (*i.e.*, $P\_leader(u)$ holds) of smallest identifier and builds a tree (of arbitrary topology) rooted at u that spans the whole connected component.

Algorithm 3: Parameters for any process u in Algorithm LEM

Inputs:

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

Ordered Magma:

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

Predicate:

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

5.2.1 Correctness of LEM. As *canBeRoot* is true for all processes, we can deduce from Theorem 3.2 (page 5) that, in a terminal configuration, $st_u = C$ for every process u. So, Algorithm LEM self-stabilizes to a terminal legitimate configuration that satisfies the following requirement:

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

One can establish (see Lemma 18 in the technical report online [22]) that in a terminal legitimate configuration of Algorithm LEM, each process u satisfies one of the following conditions:

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

5.2.2 Move Complexity of LEM. During a *GC*-segment, a process can only execute $\mathbf{R_U}$ to improve its ID. Since there are $n_{\mathtt{maxCC}}$ initial values and $n_{\mathtt{maxCC}}$ real IDs in its connected component, the total number of moves (and steps) during any execution is bounded by $(2n_{\mathtt{maxCC}}+4)(n_{\mathtt{maxCC}}+1)n$ (Theorem 4.2, page 7) *i.e.*, $O({n_{\mathtt{maxCC}}}^2 n)$.

5.3 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 STlC with the parameters given in Algorithm 4. Algorithm RSP computes (in a self-stabilizing manner) a shortest-path tree spanning the connected component of G containing r. Moreover, in any other component, the processes eventually detect the absence of r by taking the status I (Isolated).

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

Algorithm 4: Parameters for any process u in Algorithm RSP

Inputs:

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

Ordered Magma: the same as the configuration of Algorithm Forest (Algorithm 2)

Predicate:

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

5.3.1 Correctness of RSP. It can be proven that in a terminal configuration of Algorithm RSP, r is the unique process satisfying P_root. By the definition of $P\_nodeImp(u)$, we obtain the following result.

Observation 4. *In a legitimate configuration of Algorithm* RSP*, each process u satisfies one of the following three conditions:*

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

5.3.2 Move Complexity of RSP. All edges have a positive integer weight bounded by $\mathtt{W}_{\max}$, so the total number of moves (and steps) during any execution is bounded by $(\mathtt{W}_{\max}(n_{\mathtt{maxCC}}-1)^2+5)(n_{\mathtt{maxCC}}+1)n$ (by Theorem 4.5), page 8), *i.e.*, $O(\mathtt{W}_{\max} {n_{\mathtt{maxCC}}}^3 n)$.

5.4 Leader Election and Breadth-First Search Tree

Assuming the processes have unique identifiers, Algorithm LEM_BFS is the instantiation of STlC with the parameters given in Algorithm 5. In each connected component, Algorithm LEM_BFS elects the process u (*i.e.*, $P\_leader(u)$ holds) of smallest identifier and builds a breadth-first search (BFS) tree rooted at u that spans the whole connected component.

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

Algorithm 5: Parameters for any process u in Algorithm LEM_BFS

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

Ordered Magma:
the same as the configuration of Algorithm LEM (Algorithm 3)

Predicates:

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

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

Observation 5. *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.*

5.4.2 Move Complexity of LEM_BFS. All edges have the same weight, so the total number of moves (and steps) during any execution is bounded by $((n_{\mathtt{maxCC}}-1)^2+5)\cdot(n_{\mathtt{maxCC}}+1)\cdot n$ (see Corollary 5 in the technical report online [22]), *i.e.*, $O({n_{\mathtt{maxCC}}}^3 \cdot n)$.

6 CONCLUSION

We proposed a general scheme, called Algorithm STlC, to compute spanning-tree-like data structures on arbitrary (not necessarily connected) bidirectional networks. Algorithm STlC 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 $4n_{\mathtt{maxCC}}$ rounds, where $n_{\mathtt{maxCC}}$ is the maximum number of processes in a connected component. Finally, we showed that its stabilization times in steps and process moves are polynomial in usual cases.

We illustrated the versatility of our approach by presenting several instantiations of STlC that *efficiently* solve various classical problems, *i.e.*, in a linear number of rounds and polynomial number of steps and process moves. For example, assuming the processes have unique identifiers, we proposed two efficient instances of STlC 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 efficiently compute shortest-path trees for example. Assuming now an input set of roots, we also proposed an instance to efficiently compute a spanning forest of arbitrary shaped trees, with non-rooted components

detection. Again, one can easily enforce this latter construction to efficiently compute BFS or shortest-path forests. Finally, assuming a rooted network, we proposed an efficient shortest-path spanning tree construction, with non-rooted components detection. Again, efficient 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] Y. Afek and G. M. Brown. 1993. Self-Stabilization Over Unreliable Communication Media. *Distributed Computing* 7, 1 (1993), 27–34.

[2] K. Altisen, A. Cournier, S. Devismes, A. Durand, and F. Petit. 2017. Self-stabilizing Leader Election in Polynomial Steps. *Information and Computation* 254 (2017), 330–366.

[3] Anish Arora, Mohamed G. Gouda, and Ted Herman. 1990. Composite Routing Protocols. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, SPDP 1990*. IEEE Computer Society, Dallas, Texas, USA, 70–78.

[4] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. 2001. Cross-Over Composition - Enforcement of Fairness under Unfair Adversary. In *Self-Stabilizing Systems, 5th International Workshop, WSS 2001 (Lecture Notes in Computer Science)*, Ajoy Kumar Datta and Ted Herman (Eds.), Vol. 2194. Springer, Lisbon, Portugal, 19–34.

[5] Lélia Blin, Alain Cournier, and Vincent Villain. 2003. An Improved Snap-Stabilizing PIF Algorithm. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003 (Lecture Notes in Computer Science)*, Shing-Tsaan Huang and Ted Herman (Eds.), Vol. 2704. Springer, San Francisco, CA, USA, 199–214.

[6] Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. 2014. On Proof-Labeling Schemes versus Silent Self-stabilizing Algorithms. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014 (Lecture Notes in Computer Science)*, Pascal Felber and Vijay K. Garg (Eds.), Vol. 8756. Springer, Paderborn, Germany, 18–32.

[7] Lélia Blin, Maria Gradinariu Potop-Butucaru, Stephane Rovedakis, and Sébastien Tixeuil. 2010. Loop-Free Super-Stabilizing Spanning Tree Construction. In *Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium, SSS 2010 (Lecture Notes in Computer Science)*, Shlomi Dolev, Jorge Arturo Cobb, Michael J. Fischer, and Moti Yung (Eds.), Vol. 6366. Springer, New York, NY, USA, 50–64.

[8] Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. 2015. Self-stabilizing (f, g)-alliances with safe convergence. *J. Parallel and Distrib. Comput.* 81-82 (2015), 11–23.

[9] Jorge Arturo Cobb and Chin-Tser Huang. 2009. Stabilization of Maximal-Metric Routing without Knowledge of Network Size. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2009*. IEEE Computer Society, Higashi Hiroshima, Japan, 306–311.

[10] Z. Collin and S. Dolev. 1994. Self-Stabilizing Depth-First Search. *Inform. Process. Lett.* 49, 6 (1994), 297–301.

[11] A. Cournier. 2009. A lower bound for the $Max + 1$ algorithm. https://home.mis.u-picardie.fr/~cournier/MaxPlusUn.pdf. Online; accessed 11 February 2009.

[12] Alain Cournier. 2010. A New Polynomial Silent Stabilizing Spanning-Tree Construction Algorithm. In *Structural Information and Communication Complexity, 16th International Colloquium, SIROCCO 2009 (Lecture Notes in Computer Science)*, Shay Kutten and Janez Zerovnik (Eds.), Vol. 5869. Springer, Piran, Slovenia, 141–153.

[13] A. Cournier, S. Devismes, F. Petit, and V. Villain. 2006. Snap-stabilizing depth-first search on arbitrary networks. *Comput. J.* 49, 3 (2006), 268–280.

[14] Alain Cournier, Stéphane Devismes, and Vincent Villain. 2005. A Snap-Stabilizing DFS with a Lower Space Requirement. In *Self-Stabilizing Systems, 7th International Symposium, SSS 2005 (Lecture Notes in Computer Science)*, Ted Herman and Sébastien Tixeuil (Eds.), Vol. 3764. Springer, Barcelona, Spain, 33–47.

[15] A. Cournier, S. Devismes, and V. Villain. 2009. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems* 4, 1 (2009), 6:1–6:27.

[16] Alain Cournier, Stephane Rovedakis, and Vincent Villain. 2011. The First Fully Polynomial Stabilizing Algorithm for BFS Tree Construction. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011 (Lecture Notes in Computer Science)*, Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy (Eds.), Vol. 7109. Springer, Toulouse, France, 159–174.

[17] A. K. Datta, S. Devismes, K. Heurtefeux, L. L. Larmore, and Y. Rivierre. 2016. Competitive self-stabilizing k-clustering. *Theoretical Computer Science* 626 (2016), 110–133.

[18] A. K. Datta, S. Gurumurthy, F. Petit, and V. Villain. 2001. Self-Stabilizing Network Orientation Algorithms In Arbitrary Rooted Networks. *Stud. Inform. Univ.* 1, 1 (2001), 1–22.

[19] A. K. Datta, L. L. Larmore, and P. Vemula. 2011. An O(n)-time Self-stabilizing Leader Election Algorithm. *J. Parallel and Distrib. Comput.* 71, 11 (2011), 1532–1544.

[20] A. K. Datta, L. L. Larmore, and P. Vemula. 2011. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science* 412, 40 (2011), 5541–5561.

[21] S. Devismes, D. Ilcinkas, and C. Johnen. 2016. Self-Stabilizing Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Polynomial Steps. In *20th International Conference on Principles of Distributed Systems, (OPODIS 2016) (LIPIcs)*, Vol. 70. Schloss Dagstuhl, Madrid, Spain, 10:1–10:16.

[22] S. Devismes, D. Ilcinkas, and C. Johnen. 2018. *Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions.* Technical Report. HAL. https://hal.archives-ouvertes.fr/hal-01667863

[23] S. Devismes and C. Johnen. 2016. Silent self-stabilizing BFS tree algorithms revisited. *J. Parallel and Distrib. Comput.* 97 (2016), 11–23.

[24] Edsger W. Dijkstra. 1974. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM* 17, 11 (1974), 643–644.

[25] Shlomi Dolev. 2000. *Self-stabilization.* MIT Press, Cambridge, MA, USA.

[26] S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. 2011. Stabilizing data-link over non-FIFO channels with optimal fault-resilience. *Inform. Process. Lett.* 111, 18 (2011), 912–920.

[27] S. Dolev, M. G. Gouda, and M. Schneider. 1999. Memory Requirements for Silent Stabilization. *Acta Informatica* 36, 6 (1999), 447–462.

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

[29] Felix C. Gärtner. 2003. *A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms.* Technical Report. Swiss Federal Institute of Technolog (EPFL).

[30] Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. 2014. Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Networks. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014 (Lecture Notes in Computer Science)*, Pascal Felber and Vijay K. Garg (Eds.), Vol. 8756. Springer, Paderborn, Germany, 120–134.

[31] C. Glacet, N. Hanusse, D. Ilcinkas, and C. Johnen. 2016. *Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Networks - extended version.* Technical Report. LaBRI, CNRS UMR 5800. https://hal.archives-ouvertes.fr/hal-01352245

[32] M. G. Gouda and T. Herman. 1991. Adaptive Programming. *IEEE Trans. Software Eng.* 17, 9 (1991), 911–921.

[33] Shing-Tsaan Huang and Nian-Shing Chen. 1992. A Self-Stabilizing Algorithm for Constructing Breadth-First Trees. *Inform. Process. Lett.* 41, 2 (1992), 109–117.

[34] A. Kosowski and L. Kuszner. 2005. 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) (Lecture Notes in Computer Science)*, Vol. 3911. Springer, Poznan, Poland, 75–82.

[35] Alberto Leon-Garcia and Indra Widjaja. 2004. *Communication Networks* (2 ed.). McGraw-Hill, Inc., New York, NY, USA.

[36] Morris Sloman and Jeff Kramer. 1987. *Distributed systems and computer networks.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[37] G Tel. Second edition 2001. *Introduction to distributed algorithms.* Cambridge University Press, Cambridge, UK.