Algorithmica manuscript No.
(will be inserted by the editor)

Optimized Silent Self-Stabilizing Scheme for Tree-based Constructions

Stéphane Devismes* · **David Ilcinkas** · **Colette Johnen**

Received: date / Accepted: date

Abstract We propose a general scheme to compute tree-based data structures on arbitrary networks. This scheme is self-stabilizing, silent, and despite its generality, 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 at most $4n_{\texttt{maxCC}}$ rounds, where $n_{\texttt{maxCC}}$ is the maximum number of processes in any connected component of the network.

We illustrate the versatility and efficiency of our approach by proposing several instantiations solving classical spanning tree problems such as DFS, BFS, shortest-path, or unconstrained spanning tree/forest constructions, as well as other fundamental problems like leader election or finding maximum-bottleneck-bandwidth paths.

We also exhibit polynomial upper bounds on its stabilization time in steps and process moves, holding for a large class of instantiations. In several cases, the polynomial step and move complexities we obtain for those instantiations match the best known complexities of existing algorithms, despite the latter being dedicated to particular problems.

* Corresponding Author.

This study has been partially supported by the French ANR projects ANR-16-CE40-0023 (DESCARTES) and ANR-16-CE25-0009 (ESTATE). A preliminary version of this work has been presented in ICDCN'2019 [28].

Stéphane Devismes
Univ. Picardie Jules Verne, France
E-mail: Stephane.Devismes@u-picardie.fr

David Ilcinkas
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR5800, F-33400 Talence, France
E-mail: david.ilcinkas@labri.fr

Colette Johnen
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR5800, F-33400 Talence, France
E-mail: Colette.Johnen@labri.fr

Furthermore, a significant set of instantiations of our scheme requires only bounded memory space per process. This set includes, but is not limited to, DFS, BFS, and shortest-path spanning tree constructions.

Keywords distributed algorithms · self-stabilization · stabilization time · space complexity · spanning tree · leader election · spanning forest

Mathematics Subject Classification (2010) MSC 68W15 · MSC 68M15

1 Introduction

A *self-stabilizing algorithm* [30] 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. Self-stabilization makes no hypotheses on the nature (*e.g.*, memory corruption or topological changes) or extent of transient faults that could hit the system. A self-stabilizing system recovers from the effects of those faults in an unified manner. Now, such versatility comes at a price, *e.g.*, after transient faults cease, there is a finite time period, called the *stabilization phase*, during which the safety properties of the system are not guaranteed. Hence, self-stabilizing algorithms are mainly compared according to their *stabilization time*, the maximum duration of the stabilization phase.

Among the vast self-stabilizing literature, many works (see [38] for a survey) focus on *tree-based 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* [32]: 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 [32], 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 tree-based 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 [47] 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 [37], fair composition [31], cross-over composition [4], and conditional composition [22]; and many self-stabilizing algorithms are actually made as a composition of a silent tree-based construction and another algorithm designed for tree/forest topologies, *e.g.* [3,7,21]. Notably, the silence property is not mandatory in such designs, however it allows to write simpler proofs [23]. Finally, notice that silent tree-based constructions have also been used to build very general results, *e.g.*, the self-stabilizing proof-labeling scheme constructions proposed in [6].

We consider here the locally shared memory model with composite atomicity introduced by Dijkstra [30,2], which is the most commonly used model in self-stabilization. In this model, executions proceed in atomic steps (in which 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 daemon assumption.

The stabilization time of self-stabilizing algorithms is usually evaluated in terms of rounds, which capture the execution time according to the speed of the slowest processes. However, another crucial issue is the number of local state updates, *i.e.*, the number of *moves*. Indeed, the stabilization time in moves captures the amount of computations an algorithm needs in order to recover a correct behavior. Notice that the number of moves and the number of (atomic) 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]

The daemon assumption and the time complexity are closely related. Indeed, to obtain practical solutions, the designer usually tries to avoid strong assumptions on the daemon, like for example, assuming that all executions are synchronous. Now, when the considered daemon does not enforce any bound on the execution time of processes, the stabilization time in moves (resp. in steps) can be bounded only if the algorithm works under an unfair daemon. For example, if the daemon is assumed to be *distributed and weakly fair* (a daemon stronger than the distributed unfair one) and the studied algorithm actually requires the weakly fairness assumption to stabilize, then it is possible to construct executions whose convergence is arbitrarily long in terms of atomic steps (and so in moves), meaning that, in such executions, there are processes whose moves do not make the system progress in the convergence. In other words, these latter processes waste computation power and so energy. Such a situation should be therefore 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.* [1,9,24,25,34]. However, analyses of the stabilization time in steps, or moves, remain rather unusual and this may be an important issue. Indeed, recently, several self-stabilizing algorithms working under a distributed unfair daemon have been shown to have an exponential stabilization time in steps in the worst case. In [1], silent leader election algorithms from [24,25] are shown to be exponential in steps in the worst case. In [29], the Breadth-First Search (BFS) algorithm of Huang and Chen [39] is also shown to be exponential in steps. Finally, in [35] authors show that the silent self-stabilizing algorithm they proposed in [34] is also exponential in steps.

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

Contribution. In this paper, we propose a general scheme to compute tree-based data structures on bidirectional weighted networks of arbitrary topology (*n.b.*, the topologies are not necessarily connected). This algorithm 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, our scheme is efficient. Indeed, its stabilization time is at most $4n_{\texttt{maxCC}}$ rounds, where $n_{\texttt{maxCC}}$ is the maximum number of processes in a connected component. Moreover, its stabilization time in moves (and so in steps) is polynomial in 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 and efficiency of our approach, we propose several instantiations for solving classical tree-based problems.

Assuming an input set of roots, we propose an instantiation to compute a spanning forest of arbitrary shaped trees, with non-rooted components detection.[2] This instantiation stabilizes in $O(n_{\texttt{maxCC}} \cdot n)$ moves, which matches the best known step complexity for spanning tree construction [13] with explicit parent pointers.[3]

Assuming then a rooted network with positive integer weights, we propose shortest-path spanning tree and DFS constructions, with non-rooted components detection. The shortest-path spanning tree construction stabilizes in $O({n_{\texttt{maxCC}}}^3 \cdot n \cdot \texttt{W}_{\max})$ moves, where $\texttt{W}_{\max}$ is the maximum weight of an edge. This move complexity matches the best known move complexity for this problem [27].

Assuming now that the network is identified (*i.e.*, processes have distinct IDs), we propose two instantiations for electing a leader in each connected component and building a spanning tree rooted at each leader. In one version, stabilizing in $O({n_{\texttt{maxCC}}}^2 \cdot n)$ moves, the trees are of arbitrary topology. This move complexity matches the best known step complexity for leader election [1]. In the other version, stabilizing in $O({n_{\texttt{maxCC}}}^3 \cdot n)$ moves, the leader is guaranteed to be the process of minimal identifier in the connected component, and trees are BFS.

Finally, assuming a rooted network with a bandwidth assigned to each edge, we propose an instantiation computing for each process the maximum bottleneck bandwidth to the root, and a corresponding path (with the fewest edges).

Besides, most of these instantiations can be set to require only bounded memory (at the price of providing to the processes some knowledge about the graph topology; typically an upper bound on the number of processes).

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

[3] Actually, there exists a solution with implicit parent pointer [43] 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.

	Spanning Tree with explicit parent pointers	Shortest-path Spanning Tree	Leader Election
Ref.	[13]	[27]	[1]
Steps	$O(n^2)$	$O(\mathtt{W}_{\max}{n_{\mathtt{maxCC}}}^3 n)$	$O(n^3)$
Rounds	$\leq 4n$	$\leq 3n_{\mathtt{maxCC}} + D$	$\leq 3n_{\mathtt{maxCC}} + D$
Instance	Forest	RSP	LE
Steps	$O(n^2)$	$O(\mathtt{W}_{\max}{n_{\mathtt{maxCC}}}^3 n)$	$O(n^3)$
Rounds	$\leq 4n$	$\leq 4n_{\mathtt{maxCC}}$	$\leq 4n$

Table 1 Comparison between the stabilization time of some instances of our scheme and solutions from the literature (to be fair, we have replaced $n_{\mathtt{maxCC}}$ by n when the paper from the literature assumes a connected network).

Furthermore, one can easily derive from these various examples other silent self-stabilizing tree-based constructions. A comparison table between some instances of our scheme and solutions from the literature is given in Table 1.

Related Work. This work is inspired by [27]. This paper also considers the composite atomicity model under the distributed unfair daemon. The proposed algorithm is efficient both in terms of rounds and moves, tolerates disconnections, but is restricted to the case of the shortest-path tree in a rooted network. Generalizing this work to obtain a generic yet efficient self-stabilizing algorithm requires a fine tuning of the algorithm (presented in Section 3) and a careful rewriting of the proofs of correctness (presented in the remaining sections). In particular, almost all the concepts used to prove termination or complexities need to be redefined to suit the new, more general setting. Consequently, their new properties and the corresponding proofs are mostly novel.

Another closely related work is the one of Cobb and Huang [11]. In that paper, a generic self-stabilizing algorithm is presented for constructing in a rooted connected network a spanning tree where a given metric is maximized. Now, since the network is assumed to be rooted (*i.e.*, a leader process is already known), leader election is not an instance of their generic algorithm. Similarly, since they assume connected networks, the non-rooted components detection cannot be expressed too. Finally, their algorithm is proven in the composite atomicity model yet assuming a strong scheduling assumption: the sequential weakly fair daemon.

General schemes for arbitrary connected and identified networks have been proposed to transform almost any algorithm (specifically, those algorithms that can be self-stabilized) into its corresponding stabilizing version [41, 8, 14, 36]. 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 [41] and [8], authors respectively consider self-stabilization in asynchronous message-passing systems and in the synchronous locally shared memory model, while expressiveness of snap-stabilization is studied in [14, 36] assuming the locally shared memory model with composite atomicity and a distributed unfair daemon.

In [33,26], the 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 [33], they consider the locally shared memory model with read/write atomicity, while in [26], they generalize their approach to asynchronous message-passing systems. In both papers, they establish a stabilization time in $O(D + |\mathrm{S}|)$ rounds, where D is the network diameter and S is the set on which the r-operator applies. However, this bound is actually proven for the synchronous case only.

The remainder of the related work only concerns the locally shared memory model with composite atomicity assuming a distributed unfair daemon. In [6], authors use the concept of labeling scheme introduced by Korman *et al.* [42] to design silent self-stabilizing algorithms with bounded memory per process. Using this approach, they show that every static task has a silent self-stabilizing algorithm which converges within a linear number of rounds in an arbitrary identified network. No step (nor move) complexity is given.

Efficient and general schemes for snap-stabilizing (non silent) waves in arbitrary connected and rooted networks are investigated in [17]. The obtained snap-stabilizing algorithms execute each wave in a polynomial number of rounds and steps.

Few other works consider the design of particular tree-based constructions and their step complexity. Self-stabilizing algorithms that construct BFS trees in arbitrary connected and rooted networks are proposed in [18,19]. The algorithm in [18] is not silent and has a stabilization time in $O(\Delta \cdot n^3)$ steps, where Δ is the maximum degree of the network. The silent algorithm given in [19] has a stabilization time in $O(D^2)$ 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 [13,43]. The solution proposed in [13] stabilizes in at most $4 \cdot n$ rounds and at most $5 \cdot n^2$ steps, while the algorithm given in [43] stabilizes in at most $n \cdot D$ moves. However, the round complexity of this latter algorithm is not analyzed, and the parent of a process is not computed explicitly. Furthermore, Cournier [20] showed that the straightforward variant of this algorithm where a parent pointer variable is added has a stabilization time in $\Omega(n^2 \cdot D)$ steps in an infinite class of networks.

Several other papers propose self-stabilizing algorithms stabilizing in both a polynomial number of rounds and a polynomial number of steps, *e.g.* [1] (for the leader election in arbitrary identified and connected networks), and [15,16] (for the DFS token circulation in arbitrary connected and rooted networks). The silent leader election algorithm proposed in [1] stabilizes in at most $3 \cdot n + D$ rounds and $O(n^3)$ steps. The DFS token circulations given in [15,16] 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 [15], 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 our general scheme. Its proof of correctness is given in Section 4. A complexity analysis in moves is presented in Section 5, whereas an analysis of the stabilization time in rounds is proposed in Section 6. Various instantiations with their specific complexity analyses are presented in Section 7. Finally, we make concluding remarks in Section 8.

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 is conveniently 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 [46]. 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* [30, 2] in which processes communicate using a finite number of locally shared registers, called *variables*. In one indivisible move, each process can read its own variables and that of its neighbors, performs local computation, and may change only 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*. By extension, 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 a so-called *daemon*; then every process of $\mathcal{X}$ *atomically* executes one of its enabled rules, leading to a new configuration γ'. The atomic transition from γ to γ' is called a *step*. We also say that each process of $\mathcal{X}$ executes an *action* or simply *moves* during the step from γ to γ'. The possible steps induce a binary relation over $\mathcal{C}$, denoted by $\mapsto$. An *execution* is a maximal sequence of configurations $e = \gamma_0\gamma_1 \cdots \gamma_i \cdots$ 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.

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

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

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

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

Three main units of measurement are used to evaluate time complexity in our model: the number of *moves*, *steps*, and *rounds*. The definition of a 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 does not execute any action 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 an action 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 (starting from any initial configuration) to reach a terminal (legitimate) configuration.

Finally, a self-stabilizing algorithm requires *bounded memory space* if there exists a finite set S such that, along any execution from any configuration in which the states belong to S, all the reached states of any process u still belong to S.

3 Algorithm TbC

3.1 The Problem

We propose a general silent self-stabilizing algorithm, called TbC (stands for Tree-based Constructions), which aims at converging to a terminal configuration where a specified spanning forest (maybe a single spanning tree) is (distributedly) defined. The various definitions used in TbC and its code are respectively summarized and formally given in Figure 1 and Algorithm 1. Furthermore, a (slightly simplified) version of TbC has been implemented in Java and some instantiations can be simulated and visualized [40]. We invite the reader to use this tool to facilitate its understanding of the paper.

In this algorithm, each process u has three[4] constant inputs.

$canBeRoot_u$: a Boolean value, which is true if u is allowed to be the 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 $\mathcal{C}$, if there is at least one candidate $u \in \mathcal{C}$, then every process of $\mathcal{C}$ should belong to a tree (so there is at least one tree root in $\mathcal{C}$) in any terminal configuration.
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 $IDs = \mathrm{N} \cup \{\perp\}$ is totally ordered by $<$ and $\min_<(IDs) = \perp$. The value of $pname_u$ is problem dependent. Actually, we consider 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*.

$distRoot_u$: a distance belonging to $DistSet$ whose description is given below. The value $distRoot_u$ is the distance value that the candidate u should take when it is a tree root; see the variable d_u below.

Every tree is based on some kind of distance. We denote by $DistSet$ the distance domain. We use distances to detect cycles. However, according to the specific problem we consider, we may also want to minimize the weight of the trees using the distances. Distances are computed using weights on arcs. Each edge $\{u, v\}$ has then two *weights* belonging to $DistSet$: $\omega_u(v)$ denotes the weight of the arc (u, v), and $\omega_v(u)$ denotes the weight of the arc (v, u). More precisely, we need an ordered magma $(DistSet, \oplus, \prec)$, *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. Predicates $P\_neighActive(u)$, $P\_rootActive(u)$, and P_toBeC used in the algorithm are also problem dependent; see Figure 1. If a process u satisfies $P\_neighActive(u)$ or $P\_rootActive(u)$, then u is required to minimize the weight of its tree. This minimization uses the ordered magma and the problem dependent constant $distRoot_u$.

[4] Actually, there might be a fourth constant input, namely the set $distSetFinite$, when it is used in the predicate P_toBeC. This point will be discussed later in the description.

We assume that, for every arc (u, v) of G and for every values d_1 and d_2 in $DistSet$, we have

- $d_1 \prec d_1 \oplus \omega_u(v)$, and
- if $d_1 \prec d_2$, then $d_1 \oplus \omega_u(v) \preceq d_2 \oplus \omega_u(v)$.

Finally, for every integer $i \geq 0$, we define $d_1 \oplus (i \cdot d_2)$ as follows:

- $d_1 \oplus (0 \cdot d_2) \equiv d_1$
- $d_1 \oplus (i \cdot d_2) \equiv (d_1 \oplus ((i-1) \cdot d_2)) \oplus d_2$ if $i > 0$.

3.2 The Variables

In TbC, 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 first two status, I and C, are involved in the normal behavior of the algorithm, while the two other ones, EB and EF, are used during the error correction. 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 $st_u = I$.

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

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

3.3 Typical Execution

Assume that 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 $\mathcal{C}$ where at least one process is a candidate. Then, any process u of status I that is a candidate or that satisfies the predicate P_toBeC (this latter case cannot occur during the first step) is enabled to execute rule $\mathbf{R_R}$. If selected by the daemon, it executes $\mathbf{R_R}(u)$ to initiate a tree or to join a tree rooted at some candidate, choosing among the different possibilities the one that minimizes its distance value. Using this rule, it also switches its status to C and sets d_u to $distRoot_u$, or to $d_v \oplus \omega_u(v)$ if it chooses a parent v. In order to more precisely describe what happens, we focus on the two possible definitions of P_toBeC; see Figure 1.

Case $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C)$. Executions of rule $\mathbf{R_R}$ are asynchronously propagated in $\mathcal{C}$ until all processes of $\mathcal{C}$ 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 $P\_neighActive(u)$, resp. $P\_rootActive(u)$, this means that u can reduce d_u by selecting another neighbor with status C as parent, resp. by becoming a root, and this reduction is required by the specification of the problem to be solved ($P\_neighActive(u)$ and $P\_rootActive(u)$ are problem dependent). If only $P\_neighActive(u)$ is satisfied, then u chooses as parent the neighbor which allows to minimize the value of d_u. In particular, a candidate u may lose its tree root condition if it finds a sufficiently suitable parent in its neighborhood. If only $P\_rootActive(u)$ is satisfied, then u becomes a root. Finally, if both predicates are satisfied, u chooses among the two possibilities the one which minimizes the new value of d_u. Hence, the system eventually reaches a terminal configuration, where a specific spanning forest (maybe a single spanning tree) is defined (in a distributed manner) in every connected component containing at least one candidate, while all processes are isolated in the other components.

Case $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C) \wedge (distNeigh(u) \in distSetFinite)$. In that case, the variable d_u is restricted to belong to $distSetFinite$, and thus the algorithm has bounded memory space. Note that we further require in this case that $P\_neighActive(u) \equiv P\_neighValid(u) \wedge P\_toBeC(u)$; see Figure 1. Therefore, rules $\mathbf{R_U}$ are always enabled to reduce the weight of the trees if choosing some neighbor as parent makes the value of d_u decrease (while keeping it in $distSetFinite$). Once the distance values are minimized, for any process u having status C in $\mathcal{C}$, we have $d_u = d_{par_u} \oplus \omega_u(par_u)$. So, the distance of any process u having status C in $\mathcal{C}$ and at depth i in its tree belongs to $\cup_{0 \leq j \leq i} distCOk(j)$. Hence, all processes of $\mathcal{C}$ have the status C, or $\mathbf{R_R}$ is enabled at some processes with a status different from C. Eventually the system reaches a terminal configuration, where every process of $\mathcal{C}$ has the status C and verifies $\neg P\_neighValid(\cdot)$.

3.4 Error Correction

Assume now that the system is in an arbitrary configuration. Inconsistencies between the states of neighboring processes are detected using Predicate P_abRoot. We call *abnormal root* any process u satisfying $P\_abRoot(u)$. Informally (see the formal definition in Algorithm 1), a process u is an *abnormal root* if u is neither a normal root (*i.e.*, $\neg P\_root(u)$ holds), 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.*, $P\_root(u)$ holds) or u points to some neighbor (*i.e.*, $par_u \in \Gamma(u)$), and the state of u is coherent w.r.t. the state of its parent. In this latter case, $u \in Children(par_u)$, *i.e.*, u is a "real" child of its parent; see Subsection 3.5 for the formal definition.

Consider a path $\mathcal{P} = u_0, \ldots, u_k$ such that, for every $0 \leq i < k$, $u_{i+1} \in Children(u_i)$. $\mathcal{P}$ is acyclic. If u_0 is either a normal or an abnormal root, then $\mathcal{P}$ is called a *branch* rooted at u_0. Let u be a root (either normal or abnormal). We define the tree $T(u)$ as the set of all processes that belong to a branch rooted at u. If u is a normal root, then $T(u)$ is said to be a *normal tree*, otherwise u is an abnormal root and $T(u)$ is said to be an *abnormal tree*. We call 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 decreasing its distance value, since, in this case, $P\_neighActive(u)$ or $P\_rootActive(u)$ must hold), 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 [44].) Hence, the idea is to freeze T, before removing it. By freezing we mean assigning to each member of the tree an error state, here EB or 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" [10, 45] 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 the rule $\mathbf{R_{EB}}$. Then, once the EB wave reaches a leaf, the leaf initiates a convergecast EF-wave using the rule $\mathbf{R_{EF}}$. Once the EF-wave reaches the abnormal root, the tree is said to be *dead*, meaning that all processes in the tree have status EF and, consequently, no other process can join it. So, the tree can be safely deleted from its abnormal root toward its leaves. There are several possibilities for the deletion depending on whether the process u to be deleted is a candidate or verifies P_toBeC. If u is a candidate and does not verify P_toBeC: u becomes a normal root by executing $\mathbf{R_R}(u)$. If u verifies P_toBeC, again the rule $\mathbf{R_R}(u)$ is executed: u tries to directly join another "alive" tree. However if u is a candidate, and becoming a normal root allows it to further minimize d_u, then it does so. If u

is not a candidate and does not verify P_toBeC, the rule $\mathbf{R_I}(u)$ is executed: u becomes isolated, but might still join another tree later.

Let u be a process belonging to an abnormal tree of which it is not the root. Let v be its parent. From the previous explanation, it follows that during the correction, $(st_v, st_u) \in \{(C,C),\ (EB,C),\ (EB,EB),\ (EB,EF),\ (EF,EF)\}$ until v resets by $\mathbf{R_R}(v)$ or $\mathbf{R_I}(v)$. Now, due to the arbitrary initialization, the status of u and v may not be coherent, in this case u is also 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 process v with par_u (*i.e.*, $par_u = v$ and $st_v = I$), then the status of u is incoherent w.r.t. its parent v, *i.e.*, u is an abnormal root.

Actually, the freezing mechanism ensures that if a process is the root of an alive abnormal tree, it is in that situation since the initial configuration; see Lemma 9, page 20. The bounded move complexity of our scheme mainly relies on this strong property.

Algorithm 1: Algorithm TbC, predicates, macros and rules for any process u.

Predicates:

- $P\_root(u) \equiv canBeRoot_u \wedge st_u = C \wedge par_u = \perp \wedge d_u = distRoot_u$
- $P\_abRoot(u) \equiv \neg P\_root(u) \wedge st_u \neq I \wedge$
  $[par_u \notin \Gamma(u)\ \vee d_u \prec d_{par_u} \oplus \omega_u(par_u) \vee (st_u \neq st_{par_u} \wedge st_{par_u} \neq EB)]$
- $P\_reset(u) \equiv st_u = EF \wedge P\_abRoot(u)$
- $P\_neighValid(u) \equiv \{\exists v \in \Gamma(u)\ \mid\ st_v = C\} \wedge distNeigh(u) \prec d_u$
- $P\_rootValid(u) \equiv canBeRoot_u \wedge distRoot_u \prec d_u$

Macro:

- $update(u)$:
  if $\Big[\Big((P\_neighActive(u) \wedge P\_rootActive(u)) \vee (st_u \neq C \wedge P\_toBeC(u))\Big)$
  $\wedge \Big(distNeigh(u) \preceq distRoot_u \vee \neg canBeRoot_u\Big)\Big]$
  $\vee [\neg P\_rootActive(u) \wedge st_u = C\Big]$
  then
  $par_u := \text{argmin}_{\{v \in \Gamma(u)\ \mid\ st_v = C\}}(d_v \oplus \omega_u(v))$;
  $d_u := d_{par_u} \oplus \omega_u(par_u)$ (which is actually $distNeigh(u)$);
  else
  $par_u := \perp$;
  $d_u := distRoot_u$;
  end if
  $st_u := C$;

Rules:

$\mathbf{R_U}(u)$:	$st_u = C \wedge (P\_neighActive(u) \vee P\_rootActive(u))$	$\rightarrow$	$update(u)$;
$\mathbf{R_{EB}}(u)$:	$st_u = C \wedge \neg P\_neighActive(u) \wedge \neg P\_rootActive(u) \wedge$ $(P\_abRoot(u) \vee (par_u \in \Gamma(u) \wedge st_{par_u} = EB))$	$\rightarrow$	$st_u := EB$;
$\mathbf{R_{EF}}(u)$:	$st_u = EB \wedge (\forall v \in Children(u) \mid st_v = EF)$	$\rightarrow$	$st_u := EF$;
$\mathbf{R_I}(u)$:	$P\_reset(u) \wedge \neg canBeRoot_u \wedge \neg P\_toBeC(u)$	$\rightarrow$	$st_u := I$;
$\mathbf{R_R}(u)$:	$(P\_reset(u) \vee st_u = I) \wedge [canBeRoot_u \vee P\_toBeC(u)]$	$\rightarrow$	$update(u)$;

Instantiation requirements:

- $(DistSet, \oplus, \prec)$**: an ordered magma (used to represent the distances)**
- **a weight assignment (from** $DistSet$**) to all edges such that:**
  $\forall d \in DistSet$ **and** $\forall \{u,v\} \in E,\ d \prec d \oplus \omega_u(v)$
  $\forall d_1, d_2 \in DistSet$ **and** $\forall \{u,v\} \in E,\ d_1 \prec d_2 \implies d_1 \oplus \omega_u(v) \preceq d_2 \oplus \omega_u(v)$
- **a choice for the predicate** P_toBeC**:**
  Either $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C)$
  or $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C) \wedge (distNeigh(u) \in distSetFinite)$
- **a predicate** $P\_neighActive(u)$ **such that:**
  - **if** $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C) \wedge (distNeigh(u) \in distSetFinite)$**,**
    we have $P\_neighActive(u) \equiv P\_neighValid(u) \wedge P\_toBeC(u)$
  - **otherwise** $P\_neighActive(u)$ **only depends on** d_u **and** $distNeigh(u)$
- $P\_rootActive(u)$ **only depends on** $canBeRoot_u$**,** d_u**, and** $distRoot_u$
- $P\_neighActive(u) \Rightarrow P\_neighValid(u)$
- $P\_rootActive(u) \Rightarrow P\_rootValid(u)$
- **Both** $P\_neighActive(u)$ **and** $P\_rootActive(u)$ **are monotone w.r.t.** d_u**. More precisely, given** d **and** $d' \succeq d$**, if the predicate is true for** $d_u = d$**, then it is also true for** $d_u = d'$ **when all the other parameters are kept unchanged.**

Definitions:

- $Children(u) = \{v \in \Gamma(u) \mid st_v \neq I \wedge par_v = u \wedge d_v \succeq d_u \oplus \omega_v(u) \wedge (st_v = st_u \vee st_u = EB)\}$
- $distNeigh(u) = \min_{\{v \in \Gamma(u)\ \mid\ st_v = C\}}(d_v \oplus \omega_u(v))$
- $distCOk(0) = \{distRoot_v \mid v \in V\}$.
- $distCOk(i+1) = \{di \oplus \omega_u(v) \mid di \in distCOk(i) \text{ and } \{u,v\} \in E\}$.
- $distSetFinite$ a finite subset of $DistSet$ containing $\bigcup_{0 \leq j \leq n-1} distCOk(j)$

Process (constant) inputs:

- $canBeRoot_u$: a Boolean value which is true if u is a candidate
- $pname_u$: name of u
- $distRoot_u$: distance of u if it is a root
- $distSetFinite$ (if used in P_toBeC)

Process variables:

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

Fig. 1 Various inputs and definitions used in Algorithm TbC, for any process u.

3.5 Definitions

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

Root, Child, and Branch.

Definition 2 (Normal and Abnormal Roots) Every process u that satisfies $P\_root(u)$ is said to be a *normal root*. Every process u that satisfies $P\_abRoot(u)$ 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 and 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 (*i.e.*, member of the set $Children(v)$, where $v = par_u$).*

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

Observation 2 *A branch depth is at most $n_{\texttt{maxCC}} - 1$. 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).*

Lemma 1 *Let $\gamma \mapsto \gamma'$ be a step. Let $v_0, \ldots, v_k$ be an illegal branch in γ such that $st_{v_0} = EB$ and $st_{v_k} \in \{EB, EF\}$. If v_0 is still an alive abnormal root in γ', then $v_0, \ldots, v_k$ is still an illegal branch such that $st_{v_0} = EB$ and $st_{v_k} \in \{EB, EF\}$ in γ'.*

Proof By definition of an illegal branch, $v_{i+1} \in Children(v_i)$ in γ, for every $i \in [0, k)$. Now, since $st_{v_0} = EB$ and $st_{v_k} \in \{EB, EF\}$ in γ, we have $st_{v_0} \cdots st_{v_k} \in EB^+EF^*$ in γ, by definition of $Children(\cdot)$. So only $\mathbf{R_{EF}}$ may be executed, by a single process v_i with $i \in [0, k]$ in $\gamma \mapsto \gamma'$. Thus, for every $i \in [0, k)$, $(st_{v_i}, st_{v_{i+1}}) \in \{(EB, EB), (EB, EF), (EF, EF)\}$ still holds in γ'. Hence, in γ', $st_{v_k} \in \{EB, EF\}$ and $v_{i+1} \in Children(v_i)$, for every $i \in [0, k)$. This means that $v_0, \ldots v_k$ is still an illegal branch in γ'. Moreover, if v_0 is still an alive abnormal root in γ', then $st_{v_0} = EB$. □

4 Correctness of TbC

Legitimate Configurations.

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

1. $P\_root(u)$, and $\neg P\_neighActive(u)$;
2. there is a process satisfying $canBeRoot$ in V_u, $st_u = C$, $par_u \in \Gamma(u)$, $d_u \succeq d_{par_u} \oplus \omega_u(par_u)$, and $\neg P\_neighActive(u) \wedge \neg P\_rootActive(u)$;
3. there is no process satisfying $canBeRoot$ in V_u and $st_u = I$.

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

4.1 Partial Correctness

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

Lemma 2 *Any legitimate configuration of* TbC *is terminal.*

Proof Let γ be a legitimate configuration of TbC 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 TbC 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 one of the first two conditions in Definition 6. This in particular means that $st_v = C$, for every $v \in V_u$. Hence, $\mathbf{R_{EF}}(v)$, $\mathbf{R_I}(v)$, and $\mathbf{R_R}(v)$ are all disabled at every $v \in V_u$ in γ. Moreover, we have $\neg P\_neighActive(u) \wedge \neg P\_rootActive(u)$ for every $v \in V_u$ (if $P\_root(v)$ is satisfied, then $P\_rootValid(v)$ and thus $P\_rootActive(v)$ are not), which implies that $\mathbf{R_U}(v)$ is disabled at every $v \in V_u$ in γ. Finally, $st_v = C \wedge [P\_root(v) \vee (par_v \in \Gamma(v) \wedge d_v \succeq d_{par_v} \oplus \omega_v(par_v))]$ for every $v \in V_u$ implies $\neg P\_abRoot(v) \wedge st_{par_v} \neq EB$ 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 TbC is enabled at any process of V_u in γ. □

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

Lemma 3 *In any terminal configuration of* TbC*, every process has status I or C.*

Proof Assume that there exists some process that has status EB. Consider a process u with status EB having the maximum distance value. Note that no process v that has status C can be a child of u, otherwise $\mathbf{R_U}(v)$ or $\mathbf{R_{EB}}(v)$

would be enabled. Therefore, by definition of $Children(u)$ and the maximality of d_u, u has only children of status EF. Thus $\mathbf{R_{EF}}(u)$ is enabled, a contradiction.

Assume now that there exists some process that has status EF. Consider a process u with status EF having the smallest distance value. As no process has status EB (see the previous case), u is an abnormal root. Now, since u is an abnormal root of status EF, $P\_reset(u)$ holds. So, either $\mathbf{R_I}(u)$ or $\mathbf{R_R}(u)$ is enabled, a contradiction. □

Lemma 4 *Let γ be any terminal configuration of* TbC *and u be any process such that $st_u = C$ in γ. Then, u satisfies $P\_root(u)$ or $par_u \in \Gamma(u) \wedge st_{par_u} = C \wedge d_u \succeq d_{par_u} \oplus \omega_u(par_u)$ in γ.*

Proof In γ, u satisfies $\neg P\_abRoot(u)$ because, otherwise, either $\mathbf{R_U}(u)$ or $\mathbf{R_{EB}}(u)$ would be enabled, as $st_u = C$ in γ. We can thus conclude by Observation 1 that u satisfies $P\_root(u)$ or $par_u \in \Gamma(u) \wedge st_{par_u} = C \wedge d_u \succeq d_{par_u} \oplus \omega_u(par_u)$ in γ. □

Corollary 1 *Let γ be a terminal configuration of* TbC *and u be any process such that $st_u = C$ and $\neg P\_neighValid(u)$ hold in γ. Then, u satisfies $P\_root(u)$ or $par_u \in \Gamma(u) \wedge d_u = d_{par_u} \oplus \omega_u(par_u)$ in γ. Moreover, if $par_u \in \Gamma(u)$ and $d_{par_u} \in distCOk(i-1)$, then $d_u \in distCOk(i)$.*

Definition 8 Let `MinimizeFinite` be any instance of Algorithm TbC where $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C) \wedge (distNeigh(u) \in distSetFinite)$.

Lemma 5 *Let γ be a terminal configuration of* `MinimizeFinite` *and u be any process such that $st_u = C$ in γ. Then, $\neg P\_neighValid(u)$ holds in γ.*

Proof Assume that one or several processes having status C verify the predicate $P\_neighValid(\cdot)$ in γ. Let us consider such a process u that has the smallest distance d_u.

We have $P\_neighActive(u) \equiv P\_neighValid(u) \wedge P\_toBeC(u)$. As γ is terminal, $P\_neighActive(u)$ is not verified : hence $P\_toBeC(u)$ is not verified. By definition of u, a neighbor of u, named v, verifies $d_v \oplus \omega_u(v) = distNeigh(u) \prec d_u$, and $st_v = C$. Let $v_0, \cdots, v_k = v$ be the branch whose terminal extremity is v (*n.b.*, by definition, for $0 < i \leq k$, $v_{i-1} \in \Gamma[v_i]$). We have $d_{v_i} \prec d_u$, for $0 \leq i \leq k$; so $\neg P\_neighValid(v_i)$ is verified. According to Corollary 1, $P\_root(v_0)$ is verified and for $0 \leq i \leq k$, we have $d_{v_i} \in distCOk(i)$, Moreover u is not in this branch, so $k \leq n-2$. We conclude that $distNeigh(u) \in distSetFinite$, leading to a contradiction: $P\_toBeC(u)$ is verified. □

By the Lemma 5 and Corollary 1, we obtain the following result.

Corollary 2 *Let γ be any terminal configuration of* `MinimizeFinite` *and u be any process such that $st_u = C$ in γ. If u is at depth k, then $d_u \in distCOk(k)$.*

Lemma 6 *Let γ be a terminal configuration of* TbC *and u be a process such that V_u contains at least one process satisfying canBeRoot. In γ, u satisfies:*

- $st_u = C$,
- $\neg P\_neighActive(u)$ *and* $\neg P\_rootActive(u)$, *and*
- $P\_root(u)$, *or* $par_u \in \Gamma(u) \wedge d_u \succeq d_{par_u} \oplus \omega_u(par_u)$.

Proof Let v be a process of V_u such that $canBeRoot_v$ in γ. We have $st_v \in \{C, I\}$, by Lemma 3. Now, $st_v \neq I$, because otherwise $\mathbf{R_R}(v)$ would be enabled in γ. Therefore $st_v = C$ in γ.

Assume then, by contradiction, that there exists some process of V_u that has status I in γ. Consider now a process w of V_u such that w has status I and at least one of its neighbors has status C in γ (such a process exists because every process has status I or C in γ, by Lemma 3, whereas at least one process, *e.g.* v, of V_u has status C, and V_u is connected). According to the definition of $P\_toBeC(w)$, we have two cases:

Case 1: $P\_toBeC(w) \equiv (\exists v \in \Gamma(w) \mid st_v = C)$.
$\mathbf{R_R}(w)$ would be enabled in γ, a contradiction.

Case 2: $P\_toBeC(w) \equiv (\exists v \in \Gamma(w) \mid st_v = C) \wedge (distNeigh(w) \in distSetFinite)$.
Any neighbor w' of w that has the status C is at depth smaller than $n-1$ in their legal branch. Therefore, $d_{w'}$ belongs to $\bigcup_{0 \leq j \leq n-2} distCOk(j)$ by Corollary 2, and thus $distNeigh(w) \in distSetFinite$. Hence, $\mathbf{R_R}(w)$ is enabled in γ, a contradiction.

Hence, we conclude that every process of V_u (including u) has status C in γ.

Moreover, since $st_u = C$ in γ, $\neg P\_neighActive(u)$ and $\neg P\_rootActive(u)$ hold in γ (otherwise, $\mathbf{R_U}(u)$ would be enabled).

Finally, Lemma 4 allows us to conclude the proof. □

Lemma 7 *Let γ be a terminal configuration of* TbC *and u be a process such that V_u contains no process satisfying canBeRoot in γ. In γ, $st_u = I$.*

Proof Assume, for the purpose of contradiction, that u is not isolated in γ. By Lemma 3, u has status C in γ. Without loss of generality, assume u is a process subject to that condition with the smallest distance d_u in γ. Then, u is an abnormal root and enabled to execute $\mathbf{R_{EB}}(u)$ or $\mathbf{R_U}(u)$ in γ, a contradiction. □

Therefore, by Lemmas 6 and 7, we obtain the following result.

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

4.2 Bounded Memory Space

Consider `MinimizeFinite`. By definition, $distSetFinite$ is a finite set. Then, there are two rules that allow to update the variable d_u: $\mathbf{R_U}(u)$ and $\mathbf{R_R}(u)$. These rules use the macro $update(u)$ to compute the new value of d_u. If d_u

takes value $distRoot_u$, then d_u trivially remains in $distSetFinite$. Otherwise, we should remark that $P\_neighActive(u) \Rightarrow P\_toBeC(u)$ and if the executed rule is $\mathbf{R_R}(u)$, then $st_u \neq C$. Consequently, $P\_toBeC(u)$ holds, which implies that d_u also remains in $distSetFinite$ in this case. Hence, we obtain the following result.

Observation 3 `MinimizeFinite` *only requires bounded memory space.*

Moreover, such instantiations have terminal configurations which are essentially the same as those using unbounded memory space. More precisely, by Definition 6, Lemmas 2 and 5, and Theorem 1, we have the following property.

Corollary 3 *Consider any instantiation $\mathcal{I}$ identical to* `MinimizeFinite` *except that $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C)$ and $P\_neighActive(u) \equiv P\_neighValid(u)$.*

In `MinimizeFinite`*, we have $P\_neighActive(u) \equiv P\_neighValid(u) \wedge P\_toBeC(u)$, so* `MinimizeFinite` *and $\mathcal{I}$ have the same terminal configurations.*

In other words, any instantiation requiring unbounded memory space such that $P\_neighActive(u) \equiv P\_neighValid(u)$ can be turned into a bounded memory space version with the same terminal configurations and, as we will see, the same upper bounds on the number of steps and rounds.

5 Step Complexity of TbC

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

5.1 $\mathcal{C}$-Segments

Definition 9 **(AAR)** Let $\mathcal{C}$ be a connected component of G and let γ be a configuration. $AAR(\gamma, \mathcal{C})$ is the set of processes $u \in \mathcal{C}$ such that, in γ, u is an alive abnormal root, or $P\_rootActive(u) \wedge st_u = C$ holds.

We now prove that $AAR(\gamma, \mathcal{C})$ can never gain any new element.

Lemma 8 *Let $\gamma \mapsto \gamma'$ be a step where a process u executes the rule $\mathbf{R_U}$ or $\mathbf{R_R}$. Then, u is not an alive abnormal root in γ'.*

Proof We have $par_u \in \Gamma(u) \cup \{\perp\}$ in γ'. We separately deal with these two cases.

First assume that $par_u = \perp$ in γ'. We claim that, in this case, the constant $canBeRoot_u$ is true. Recall that $P\_rootActive(u)$ implies $P\_rootValid(u)$,

which in turn implies $canBeRoot_u$. Thus, for the purpose of contradiction, assume that neither $canBeRoot_u$ nor $P\_rootActive(u)$ hold. Note that $par_u = \bot$ in γ' implies that u must have executed the **else** part of $update(u)$ in $\gamma \mapsto \gamma'$. Therefore, the second part of the main disjunction of the **if** condition must not hold. This means that $st_u \neq C$, and thus that u has executed the rule $\mathbf{R_R}$ in $\gamma \mapsto \gamma'$. For this rule to be enabled, the second part of the conjunction in its guard must hold, which implies that $P\_toBeC(u)$ holds. However, this last property, combined with the preceding ones, validates the **if** condition in $update(u)$, which contradicts $par_u = \bot$ in γ'. Therefore, $canBeRoot_u$ is indeed true and thus $P\_root(u)$ is true in γ', which, in turn, implies $\neg P\_abRoot(u)$.

Assume now that $par_u = v \in \Gamma(u)$ in γ'. Then $st_v = C$ in γ, because par_u is chosen in the set $\{v \in \Gamma(u) \mid st_v = C\}$; see $update(u)$. Consequently, the only rules that v may execute in $\gamma \mapsto \gamma'$ are $\mathbf{R_U}$ or $\mathbf{R_{EB}}$. In $\gamma \mapsto \gamma'$, v either takes the status EB, decreases its distance value, or does not change the value of its variables. In all cases, u belongs to $Children(v)$ in γ', which prevents u from being an alive abnormal root in γ', by Observation 1. More precisely, if v decreases its distance value, the fact that u still belongs to $Children(v)$ in γ' comes from the hypothesis on $\oplus$ stating that for any distances d_1 and d_2 and any weight ω of an edge, we have $d_1 \oplus \omega \preceq d_2 \oplus \omega$ if $d_1 \prec d_2$. □

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

Lemma 9 *No alive abnormal root is created along any execution of* TbC.

Proof Let $\gamma \mapsto \gamma'$ be a step and u be a process that is not an alive abnormal root in γ. Assume, by contradiction, that u is an alive abnormal root in γ'.

If the status of u is EF or I in γ', then u is not an alive abnormal root in γ'. If u executes $\mathbf{R_U}$ or $\mathbf{R_R}$ during this step, then u is not an alive abnormal root in γ' either, by Lemma 8. So the only rule that u may execute is $\mathbf{R_{EB}}$ in $\gamma \mapsto \gamma'$. Furthermore, both in γ and γ', u has status C or EB, and $par_u \in \Gamma(u) \cup \{\bot\}$ (because u is not an abnormal root in γ).

Assume first that $par_u = \bot$ in γ'. Then, $par_u = \bot$ already holds in γ. We thus have $P\_root(u)$ in γ because $\neg P\_abRoot(u)$ in γ. Consequently, u executes no move in $\gamma \mapsto \gamma'$, and u is still a normal root in γ', a contradiction.

Assume now that $par_u = v \in \Gamma(u)$ in γ'. Whether u executes $\mathbf{R_{EB}}$ or not, par_u already designates v in γ. Also, $\neg P\_abRoot(u)$ in γ implies that $u \in Children(v)$ and $st_v \in \{C, EB\}$ in γ, further implying that the only rules that v may execute in $\gamma \mapsto \gamma'$ are $\mathbf{R_U}$ or $\mathbf{R_{EB}}$. Moreover, if $st_u = EB$ in γ or u executes $\mathbf{R_{EB}}$ in $\gamma \mapsto \gamma'$, then $st_v = EB$ too in γ and v does not move in $\gamma \mapsto \gamma'$. Consequently, $\neg P\_abRoot(u)$ still holds in γ', a contradiction. Otherwise, $st_u = C$ in γ and u does not move in $\gamma \mapsto \gamma'$. In $\gamma \mapsto \gamma'$, v either takes the status EB, decreases its distance value, or does not change the value of its variables. In all cases, u still belongs to $Children(v)$ in γ', which prevents u from being an alive abnormal root in γ' (by Observation 1), a contradiction. □

Lemma 10 *If a process u satisfies $P\_rootActive(u) \wedge st_u = C$, then it does so, and it never performed any move, since the beginning of the execution.*

Proof Let u be a process satisfying $P\_rootActive(u) \wedge st_u = C$. Note that the property does only depend on the local state on u. Assume thus, for the purpose of contradiction, that u did perform at least a move since the beginning of the execution, and consider the last such move, say during the step $\gamma \mapsto \gamma'$.

Since $st_u = C$ holds in γ', u must have executed $\mathbf{R_U}$ or $\mathbf{R_R}$ during that step. For $P\_rootActive(u)$ to hold in γ', u must have executed the **then** part of $update(u)$; otherwise $d_u = distRoot_u$ in γ' and so $\neg P\_rootValid(u)$, which implies $\neg P\_rootActive(u)$.

If it did so because of the first part of the disjunction in the **if** of $update(u)$, then $distNeigh(u) \preceq distRoot_u$, which implies that $d_u \preceq distRoot_u$ in γ'. Thus, in that case, $P\_rootActive(u)$ cannot hold in γ', a contradiction.

Otherwise (it did so because of the second part of the disjunction), in γ, $st_u = C$ holds but $P\_rootActive(u)$ does not. During $\gamma \mapsto \gamma'$, u executed $\mathbf{R_U}$ and decreased its distance d_u. As $P\_rootActive(u)$ is monotone w.r.t. d_u, $\neg P\_rootActive(u)$ in γ implies $\neg P\_rootActive(u)$ in γ', leading to a contradiction. □

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

Corollary 4 *For every step $\gamma \mapsto \gamma'$, $AAR(\gamma', \mathcal{C}) \subseteq AAR(\gamma, \mathcal{C})$.*

Based on Corollary 4, we can use the notion of *$\mathcal{C}$-segment* defined below to bound the total number of moves in an execution.

Definition 10 ($\mathcal{C}$-Segment) Let $e = \gamma_0\gamma_1\cdots$ be an execution of TbC and $\mathcal{C}$ be a connected component of G.

- If there is no step $\gamma_i \mapsto \gamma_{i+1}$ in e such that $|AAR(\gamma_i, \mathcal{C})| > |AAR(\gamma_{i+1}, \mathcal{C})|$, then the *first $\mathcal{C}$-segment* of e is e itself and there is no other $\mathcal{C}$-segment.
- Otherwise, let $\gamma_i \mapsto \gamma_{i+1}$ be the first step of e such that $|AAR(\gamma_i, \mathcal{C})| > |AAR(\gamma_{i+1}, \mathcal{C})|$. The *first $\mathcal{C}$-segment* of e is the prefix $\gamma_0 \cdots \gamma_{i+1}$. The *second $\mathcal{C}$-segment* of e is the first $\mathcal{C}$-segment of the suffix $\gamma_{i+1}\gamma_{i+2}\cdots$, and so forth.

By Corollary 4, and since by definition $|AAR(\gamma_i, \mathcal{C})| \leq n_{\texttt{maxCC}}$ for every connected component $\mathcal{C}$ and every configuration γ_i, we have:

Observation 4 *Let $\mathcal{C}$ be a connected component of G. For every execution e of TbC, e contains at most $n_{\texttt{maxCC}} + 1$ $\mathcal{C}$-segments.*

We now prove some properties on the moves made by a process in a $\mathcal{C}$-segment.

Lemma 11 *Let $\mathcal{C}$ be a connected component of G, u be any process of $\mathcal{C}$, and $\mathcal{S}$ be a $\mathcal{C}$-segment. During $\mathcal{S}$, if u executes the rule $\mathbf{R_{EF}}$, then u does not execute any other rule in the remaining of $\mathcal{S}$.*

Proof Let $\gamma_1 \mapsto \gamma_2$ be a step of $\mathcal{S}$ in which u executes $\mathbf{R_{EF}}$. Note that u has status EB in γ_1. Let $\gamma_3 \mapsto \gamma_4$ be the next step in which u executes a rule. (If one of these two steps does not exist, then the lemma trivially holds.)

Let v be the root (at depth 0) of any branch in γ_1 containing u. By Definition 4, v must have status EB (and so satisfies $\neg P\_rootActive(u) \vee st_u \neq C$ forever, by Lemma 10), and must therefore be an alive abnormal root. This implies that $v \in AAR(\gamma_1, \mathcal{C})$. Note that we may have $v = u$. On the other hand, in γ_3, u is the dead abnormal root of all branches it belongs to since $st_u = EF$ in γ_3 and u necessarily executes $\mathbf{R_I}$ or $\mathbf{R_R}$ in this step. By Observation 1, either $u = v$ or u no more belongs to a branch whose initial extremity is v. In either case, v is no more an alive abnormal root in γ_3. Indeed, in the first case, $u = v$ has status EF, while in the second case, if v were still an alive abnormal root, then u would still be in a branch with v as initial extremity, by Lemmas 1 and 9. Therefore $v \notin AAR(\gamma_3, \mathcal{C})$. Consequently, the steps $\gamma_1 \mapsto \gamma_2$, and $\gamma_3 \mapsto \gamma_4$ belong to two distinct $\mathcal{C}$-segments of the execution, by Corollary 4 and Definition 10. □

By Lemma 11 and from the code of the algorithm, we obtain the following result.

Corollary 5 *Let $\mathcal{C}$ be a connected component of G and u be any process of $\mathcal{C}$. The sequence of rules executed by u during a $\mathcal{C}$-segment belongs to the following language:*

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

By Observation 4 and Corollary 5, we obtain the following result.

Theorem 2 *If $\#U$ is an upper bound on the number of rules $\mathbf{R_U}$ executed by any process of $\mathcal{C}$ in any $\mathcal{C}$-segment for any connected component $\mathcal{C}$, then the* total *number of moves in any execution is bounded by $(\#U+4)\cdot(n_{\texttt{maxCC}}+1)\cdot n$.*

5.2 Causal Chains

We now use the notion of *causal chain* defined below to further analyze the number of moves and steps in a $\mathcal{C}$-segment.

Definition 11 (Causal Chain) Let $\mathcal{C}$ be a connected component of G, v_0 be a process of $\mathcal{C}$, and $\mathcal{S}$ be any $\mathcal{C}$-segment. A *causal chain of $\mathcal{S}$* rooted at v_0 is a non-empty sequence of actions $a_1, a_2, \ldots, a_k$ executed in $\mathcal{S}$ such that the action a_1 sets par_{v_1} to v_0 and for all $2 \leq i \leq k$, the action a_i sets par_{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 $\mathcal{C}$ be a connected component of G, v_0 be a process of $\mathcal{C}$, and $\mathcal{S}$ be any $\mathcal{C}$-segment. Let $a_1, a_2, \ldots, a_k$ be a causal chain of $\mathcal{S}$ 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 update(v_i) (i.e., v_i executes the rule $\mathbf{R_U}$ or $\mathbf{R_R}$), and v_i is a process of $\mathcal{C}$.*

– *Assume a_1 is executed in the step $\gamma \mapsto \gamma'$ of $\mathcal{S}$. Denote by ds_0 the distance value of process v_0 in γ; we call this value the* initiating value *of the causal chain. For all $1 \leq i \leq k$, a_i sets d_{v_i} to $((ds_0 \oplus \omega_{v_1}(v_0)) \oplus \cdots) \oplus \omega_{v_i}(v_{i-1})$.*

Lemma 12 *Let $\mathcal{C}$ be a connected component of G and $\mathcal{S}$ be a $\mathcal{C}$-segment. All actions in a causal chain of $\mathcal{S}$ are executed by different processes of $\mathcal{C}$, none of them being the root of the causal chain.*

Proof Assume, by contradiction, that there exists a process v such that, in some causal chain $a_1, a_2, \ldots, a_k$ of $\mathcal{S}$, 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 $j > i$. Process v has status C in γ_i, and the value of d_v is strictly larger in γ_{j+1} than in γ_i (by Observation 5, second item). However, any rule $\mathbf{R_U}$ executed by v makes the value of d_v decrease. Finally, since Process v has status C in γ_i, $\mathbf{R_R}(v)$ will be no more executed in $\mathcal{S}$ from that configuration, by Corollary 5 and from the rules' guards. Hence, we obtain a contradiction (by Observation 5, first item). □

5.3 Maximal Causal chains

Definition 12 (Maximal causal chain) Let $\mathcal{C}$ be a connected component of G, v_0 be a process of $\mathcal{C}$, and $\mathcal{S}$ be any $\mathcal{C}$-segment. A *maximal causal chain of* $\mathcal{S}$ rooted at v_0 is a causal chain $a_1, a_2, \ldots, a_k$ of $\mathcal{S}$ such that the causal chain is maximal and, either v_0 is a normal root or the action a_1 sets par_{v_1} to v_0 not later than any action executed by v_0 in $\mathcal{S}$.

The following lemma adds a property to Observation 5 for the specific case of maximal causal chains.

Lemma 13 *Given any connected component $\mathcal{C}$, any $\mathcal{C}$-segment $\mathcal{S}$, and any process $v \in \mathcal{C}$, all maximal causal chains of $\mathcal{S}$ rooted at v have the same initiating value.*

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

Since d_1 is the distance value of v at the beginning of $\mathcal{S}$, there must exist an action a executing the **else** part of $update(v)$ in $\mathcal{S}$. Moreover, since the maximal causal chain of $\mathcal{S}$ rooted at v with initiating value d_1 exists, $st_v = C$ initially (otherwise, no neighbor can choose v as parent before any action of v in $\mathcal{C}$). Thus, by Corollary 5, the action a is an execution of $\mathbf{R_U}$ in the case when $P\_rootActive(v) \wedge st_v = C$ holds. By definition of a $\mathcal{C}$-segment, Lemmas 8-10, and Corollary 4, the action a is thus executed during the last step of $\mathcal{S}$ and thus no maximal causal chains of $\mathcal{S}$ (which are never empty by definition)

can be rooted at v with initiating value $d_2 = distRoot_v$. This contradiction concludes the proof. □

Definition 13 ($SI_{\mathcal{S},v}$) Let $\mathcal{C}$ be a connected component of G, v be a process of $\mathcal{C}$, and $\mathcal{S}$ be a $\mathcal{C}$-segment. We define $SI_{\mathcal{S},v}$ as the set of all the distance values obtained after executing an action belonging to the maximal causal chains of $\mathcal{S}$ rooted at v.

The following lemma will be used to establish the termination of TbC in any case. It will also lead to a huge upper bound on the move complexity. However, we will see that in many practical cases, the upper bound in moves can be refined to be polynomial, see Theorem 3 and Corollary 8.

Lemma 14 *Let $\mathcal{C}$ be a connected component of G, v_0 be a process of $\mathcal{C}$, and $\mathcal{S}$ be a $\mathcal{C}$-segment. The size of the set $SI_{\mathcal{S},v_0}$ is bounded by $(n_{\texttt{maxCC}} - 1)!$ (the factorial of $n_{\texttt{maxCC}} - 1$).*

Proof Let us consider a distance value d obtained after executing an action a_i belonging to a maximal causal chain $a_1, a_2, \ldots, a_k$ of $\mathcal{S}$ rooted at v_0. Denote by v_i the process that executes a_i, for all $i \in \{1, \ldots, k\}$. By Observation 5, we have $d = ((ds_0 \oplus \omega_{v_1}(v_0)) \oplus \cdots) \oplus \omega_{v_i}(v_{i-1})$, with ds_0 being the initiating value common to all maximal causal chains of $\mathcal{S}$ rooted at v_0 (see Lemma 13). In other words, the value d is fully determined by the sequence of processes $v_1, \ldots, v_i$ (v_0 and $\mathcal{S}$ being fixed). Moreover, note that all the v_j, $0 \leq j \leq i$ are different processes and v_0 does not execute any action of any causal chain it is the root of, by Lemma 12. Therefore, $|SI_{\mathcal{S},v_0}|$ is bounded by $(n_{\texttt{maxCC}} - 1)!$. □

5.4 Move Complexity of TbC

Lemma 15 *Let $\mathcal{C}$ be a connected component of G, $u \in \mathcal{C}$, and $\mathcal{S}$ be a $\mathcal{C}$-segment. If the size of $SI_{\mathcal{S},v}$ is bounded by X for any process $v \in \mathcal{C}$, then the number of $\mathbf{R_U}$ moves done by u in $\mathcal{S}$ is bounded by $X \cdot (n_{\texttt{maxCC}} - 1) + 1$.*

Proof By Corollary 5, $\mathbf{R_U}(u)$ executions in $\mathcal{S}$ are not interrupted by the executions of other rules at u, and they make the value of d_u decrease. Therefore, all the values of d_u obtained by the $\mathbf{R_U}$ executions done by u in $\mathcal{S}$ are different. By Definitions 12 and 13, all these values belong to the set $\bigcup_{v \in \mathcal{C} \setminus \{u\}} SI_{\mathcal{S},v} \cup \{distRoot_u\}$, which has size at most $X \cdot (n_{\texttt{maxCC}} - 1) + 1$. □

By Theorem 2 and Lemma 15, we obtain the following result.

Corollary 6 *If the size of $SI_{\mathcal{S},v}$ is bounded by X for any connected component $\mathcal{C}$, any process $v \in \mathcal{C}$, and any $\mathcal{C}$-segment $\mathcal{S}$, then the total number of moves during any execution is bounded by $(X \cdot (n_{\texttt{maxCC}} - 1) + 5) \cdot (n_{\texttt{maxCC}} + 1) \cdot n$.*

Combined with Lemma 14, this corollary already allows us to prove that TbC always terminates and has a bounded move complexity.

Corollary 7 *Algorithm* TbC *is silent self-stabilizing under the distributed unfair daemon and has a bounded move (and step) complexity, a valid bound being* $5n \cdot (n_{\texttt{maxCC}} + 1)!$.

Let $\texttt{W}_{\max} = \max\{\omega_u(v) \mid u \in V \wedge v \in \Gamma(u)\}$. If all weights are strictly positive integers and $\oplus$ is the addition operator, then the size of any $SI_{\mathcal{S},u}$ is bounded by $\texttt{W}_{\max}(n_{\texttt{maxCC}}-1)$ for every connected component $\mathcal{C}$, every $\mathcal{C}$-segment $\mathcal{S}$, and every process $u \in \mathcal{C}$ because, by Observation 5 and Lemma 12, $SI_{\mathcal{S},u} \subseteq [ds_{\mathcal{S},u}+1, ds_{\mathcal{S},u}+\texttt{W}_{\max}(n_{cc}-1)]$, where $n_{cc} \leq n_{\texttt{maxCC}}$ is the number of processes in $\mathcal{C}$, and $ds_{\mathcal{S},u}$ is the common (by Lemma 13) initiating value of the maximal causal chains of $\mathcal{S}$ rooted at u. Hence, we deduce the following theorem from Corollaries 6 and 7.

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

Lemma 16 *Let* $\mathcal{C}$ *be a connected component of* G, $v \in \mathcal{C}$, *and* $\mathcal{S}$ *be a* $\mathcal{C}$*-segment. If all edges have the same weight, then* $|SI_{\mathcal{S},v}| < n_{\texttt{maxCC}}$.

Proof Assume that all edges have the same weight ω. According to Observation 5 and Lemma 12, we have $SI_{\mathcal{S},v} \subseteq \{ds_{\mathcal{S},v} \oplus (i.\omega) \mid 1 \leq i \leq n_{\texttt{maxCC}} - 1\}$, where $ds_{\mathcal{S},v}$ is the common (by Lemma 13) initiating value of the maximal causal chains of $\mathcal{S}$ rooted at v. □

By Corollary 6 and Lemma 16, we obtain the following result.

Corollary 8 *If all edges have the same weight, then the total number of moves (and steps) during any execution is bounded by* $((n_{\texttt{maxCC}}-1)^2+5)\cdot(n_{\texttt{maxCC}}+1)\cdot n$.

6 Round Complexity of TbC

6.1 Normal Configurations

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

Definition 14 (Normal Process) A process u is said to be *normal* if u satisfies the following two conditions:

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

Definition 15 (Normal Configuration) Let γ be a configuration of TbC. γ is said to be *normal* if every process is normal in γ; otherwise γ is said to be *abnormal*.

Observation 6 *In a normal configuration of* TbC*, only the rules* $\mathbf{R_U}$ *or* $\mathbf{R_R}$ *may be enabled at any process.*

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

Lemma 17 *Any step from a normal configuration of* TbC *reaches a normal configuration of* TbC*.*

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 $\neg P\_abRoot(v)$. Hence, both $\mathbf{R_{EB}}(u)$ and $\mathbf{R_{EF}}(u)$ are disabled in γ, and consequently $st_u \notin \{EB, EF\}$ still holds in γ'.

Moreover, since u is not an alive abnormal root in γ, Lemma 9 implies that u is not an alive abnormal root in γ' either. Since $st_u \neq EF$ in γ', we obtain $\neg P\_abRoot(u)$ in γ'. □

6.2 From an Arbitrary Configuration to a Normal Configuration

The lemma below essentially claims that all the processes that are in illegal branches progressively switch to status EB within $n_{\texttt{maxCC}}$ rounds, in order of increasing depth; see Definition 5, page 15, for the definition of depth.

Lemma 18 *Let* $i \in \mathbb{N}$*. From the beginning of Round* $i+1$*, there does not exist any process both in state* C *and at depth less than* i *in an illegal branch.*

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

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

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

Consider any process u with status C at depth i in an illegal branch at the beginning of Round $i+1$. It remains to prove that u will not stay so until the end of Round $i + 1$. By induction hypothesis, u is an abnormal root, or the parent of u has not status C (*i.e.*, it has status EB). During Round $i + 1$, u will execute rule either $\mathbf{R_{EB}}$ or $\mathbf{R_U}$ and thus either switch to status EB, or join another branch (at a depth greater than i if that branch is illegal), or become a normal root turning its branch to be legal. This concludes the proof of the lemma. □

Definition 16 (Almost Normal Configuration)
A configuration γ of TbC is said to be *almost normal* if in γ, every process u satisfies $st_u = C \Rightarrow \neg P\_abRoot(u) \wedge [P\_root(u) \vee (par_u \in \Gamma(u) \wedge st_{par_u} = C)]$.

Lemma 19 *Any step from an almost normal configuration of* TbC *leads to an almost normal configuration of* TbC.

Proof Let $\gamma \mapsto \gamma'$ be a step of TbC such that γ is an almost normal configuration. Assume, for the purpose of contradiction, that γ' is not an almost normal configuration. Then, by Definition 16, at least one process u satisfies one of the following two cases.

- $st_u = C \wedge P\_abRoot(u)$ in γ'. Then, Lemma 9 (page 20) implies that u is already an alive abnormal root in γ. However, since γ is an almost normal configuration, u cannot be an alive abnormal root of status C in γ. So, we necessarily have $st_u = EB$ in γ. But, in this case, $st_u \neq C$ in γ', a contradiction.
- $st_u = C \wedge \neg P\_abRoot(u) \wedge \neg P\_root(u) \wedge [par_u \notin \Gamma(u) \vee st_{par_u} \neq C]$ in γ'. Now, $\neg P\_abRoot(u) \wedge \neg P\_root(u)$ implies $par_u \in \Gamma(u)$ by Observation 1 (page 15) and thus $st_{par_u} = EB$ in γ' from $par_u \notin \Gamma(u) \vee st_{par_u} \neq C$. Let v be the process par_u in γ'. By definition of an almost normal configuration (Definition 16), v does not execute $\mathbf{R_{EB}}$ in $\gamma \mapsto \gamma'$. So, in γ, we have $st_v = EB$. If u executes $\mathbf{R_R}$ or $\mathbf{R_U}$ in $\gamma \mapsto \gamma'$ then $par_u \neq v$ in γ'. If u executes $\mathbf{R_{EB}}$, $\mathbf{R_{EF}}$ or $\mathbf{R_I}$ in $\gamma \mapsto \gamma'$ then $st_u \neq C$ in γ'. So u does not execute any rule in $\gamma \mapsto \gamma'$. Hence, we have $st_u = C$, $par_u = v$, and $st_v = EB$ in γ, meaning that γ is not an almost normal configuration, a contradiction. □

From Lemmas 18 and 19, we obtain the following corollary.

Corollary 9 *After at most $n_{\texttt{maxCC}}$ rounds, the system is in an almost normal configuration and remains so forever.*

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

Lemma 20 *Let $i \in \mathbb{N}^*$. From the beginning of Round $n_{\texttt{maxCC}} + i$, any process at depth larger than $n_{\texttt{maxCC}} - i$ in an illegal branch has status EF.*

Proof We prove this lemma by induction on i. The base case ($i = 1$) is trivial (by Observation 2, page 15), 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 $n_{\texttt{maxCC}} - i$ has status EF (by induction hypothesis). Therefore, processes with status EB at depth $n_{\texttt{maxCC}} - i$ in an illegal branch are enabled to 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 9, once Round $n_{\texttt{maxCC}}$ has terminated, any process in an illegal branch that executes some rule either gets status EF, or will be outside any illegal branch forever. □

Definition 17 (Quasi Normal Configuration) An almost normal configuration γ of TbC is said to be *quasi normal* if no process has status EB in γ.

Observation 7 *There is no alive abnormal root in a quasi normal configuration.*

Lemma 21 *Any step from a quasi normal configuration of* TbC *leads to a quasi normal configuration of* TbC*.*

Proof Let $\gamma \mapsto \gamma'$ be a step of TbC such that γ is a quasi normal configuration. First, by definition, a quasi normal configuration is also an almost normal configuration. So, γ' is almost normal (Lemma 19) and no process has status EB in γ' since no rule $\mathbf{R_{EB}}$ is enabled in γ. Hence, γ' is a quasi normal configuration. □

From Lemmas 20 and 21, we obtain the following corollary.

Corollary 10 *After at most* $2n_{\texttt{maxCC}}$ *rounds, the system is in a quasi normal configuration and remains so 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 22 *Let* $i \in \mathbb{N}^*$*. From the beginning of Round* $2n_{\texttt{maxCC}} + i$*, there does not exist any process at depth larger than* $n_{\texttt{maxCC}} - i$ *in an illegal branch.*

Proof We prove this lemma by induction on i. The base case ($i = 1$) is trivial (by Observation 2, page 15), 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 $n_{\texttt{maxCC}} - i$ in an illegal branch. All processes in an illegal branch have the status EF (by Lemma 20). So, at the beginning of Round $2n_{\texttt{maxCC}}+i$, any abnormal root satisfies the predicate P_reset, and is enabled to execute either $\mathbf{R_I}$, or $\mathbf{R_R}$. So, all abnormal roots at the beginning of 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 9, no process can join an illegal tree after $n_{\texttt{maxCC}}$ rounds have occurred. □

By Lemmas 17-22, we obtain the following result.

Theorem 4 *After at most* $3n_{\texttt{maxCC}}$ *rounds, a normal configuration of* TbC *is reached, and the configuration remains normal forever.*

6.3 From a Normal Configuration to a Terminal Configuration

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

Observation 8 *In a normal configuration of* TbC*, all processes in connected components containing no process satisfying canBeRoot are in state I and thus are disabled.*

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

From the previous observation, we only need to focus on any connected component $\mathcal{C}$ containing at least one process satisfying *canBeRoot*.

Let us fix an arbitrary execution ex of TbC in $\mathcal{C}$ starting from a normal configuration γ. By Corollary 7 (page 25), a terminal configuration is eventually reached after a finite number of steps along ex.

Lemma 23 *Let $\mathtt{ST}_{\mathcal{C}}(i, \mathsf{ex})$ be the set of processes defined as $\{u \in \mathcal{C} \mid u$ performs a move along* ex *after the beginning of Round $i\}$. If $|\mathtt{ST}_{\mathcal{C}}(i, \mathsf{ex})| > 0$ then $|\mathtt{ST}_{\mathcal{C}}(i+1, \mathsf{ex})| < |\mathtt{ST}_{\mathcal{C}}(i, \mathsf{ex})|$.*

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

Let γ_i be the configuration at the beginning of Round i of ex, and let γ_f be the terminal configuration of ex. Let us consider the process $u \in \mathtt{ST}_{\mathcal{C}}(i, \mathsf{ex})$ having the minimum distance d_u in γ_f, denoted by $dmin(i)$. According to the definition of u and Observation 8, *(*)* every process w' of $\mathtt{ST}_{\mathcal{C}}(i, \mathsf{ex})$ satisfies $dmin(i) \preceq d_{w'}$ or $st_{w'} = I$ along ex from γ_i.

Case 1. In γ_f, $par_u = \perp$.
This means that $P\_root(u)$ holds in γ_f. This further implies that, along ex from γ_i, the last action of u consists in executing the **else** part of $update(u)$. At that time, u satisfies $P\_rootActive(u) \vee st_u = I$. Actually, by Lemma 10 page 21 and Observation 8, not only this must already hold from γ_i, but this action is the unique action of u along ex. This action is thus done during the i^{th} round of ex, and $u \notin \mathtt{ST}_{\mathcal{C}}(i+1, \mathsf{ex})$, concluding the case.

Case 2. In γ_f, $par_u = w \in \Gamma(u)$.
Along ex from γ_i, u executes its last move in some step $\gamma_j \mapsto \gamma_{j+1}$. In this step, u executes the **then** part of $update(u)$ since $par_u \neq \perp$ in γ_f. In γ_j, $distNeigh(u) = dmin(i)$. By Observation 8 and *(*)*, we can conclude that $distNeigh(u)$ is constantly equal to $dmin(i)$ and $P\_toBeC(u)$ is constantly true since γ_i. From the properties of $P\_rootActive(u)$ and $P\_neighActive(u)$, this also implies that the values of those two predicates only depends on d_u from γ_i (other influencing parameters being constant from γ_i). Furthermore, since $P\_rootActive(u)$ and $P\_neighActive(u)$ are monotone w.r.t. d_u, their respective values are constant from γ_i until u moves. Assume now, by contradiction, that u moves in some step $\gamma_k \mapsto \gamma_{k+1}$ with $i \leq k < j$. Without loss of generality, assume that k is maximum. Then, u necessarily executes the **else** part of $update(u)$ in $\gamma_k \mapsto \gamma_{k+1}$ (otherwise, u is not enabled in γ_j). Thus, $distNeigh(u) \preceq distRoot_u$ so that

u is enabled in γ_j. In this case, u necessarily has status C in γ_k (u otherwise executes the **then** part of $update(u)$). Again to execute the **else** part of $update(u)$ during $\gamma_k \mapsto \gamma_{k+1}$, we should have $P\_rootActive(u)$ and $\neg P\_neighActive(u)$ in γ_k. Overall, from γ_k to γ_j, we have $st_u = C$; and from γ_{k+1} to γ_j we have $d_u = distRoot_u$, which implies $\neg P\_rootActive(u)$. Moreover, $\neg P\_neighActive(u)$ holds in γ_j since it is monotone w.r.t. d_u and by Observation 8. So, u is disabled in γ_j, a contradiction. Hence, the only move of u from γ_i is during the step $\gamma_j \mapsto \gamma_{j+1}$ and u was enabled since γ_i, *i.e.*, u executes its last move during Round i, which means that $u \notin \mathtt{ST}_{\mathcal{C}}(i+1, \mathsf{ex})$, concluding the case. □

From the previous lemma, and Theorems 1 and 4, we obtain the following result.

Corollary 11 *A terminal legitimate configuration of any instantiation of* TbC *is reached in at most* $4n_{\mathtt{maxCC}}$ *rounds from any configuration.*

7 Instantiations

In this section, we illustrate the versatility of Algorithm TbC by proposing several instantiations that solve various classical problems.

By Definition 7 and Theorem 1 (pages 16 and 18, respectively), any process u in a terminal configuration of an instance of TbC satisfies one of the three following properties.

Property 1: $P\_root(u)$ and $\neg P\_neighActive(u)$.
Property 2: There is a process satisfying *canBeRoot* in V_u, $st_u = C$, $par_u \in \Gamma(u)$, $d_u \succeq d_{par_u} \oplus \omega_u(par_u)$, and $\neg P\_neighActive(u) \wedge \neg P\_rootActive(u)$.
Property 3: There is no process satisfying *canBeRoot* in V_u and $st_u = I$.

By Corollaries 7 and 11 (pages 25 and 30, respectively), all instances of TbC reach under the unfair daemon a terminal configuration in at most $4n_{\mathtt{maxCC}}$ rounds, starting from an arbitrary one.

Observation 9 *Let $\mathcal{C}$ be a connected component of G containing a process satisfying canBeRoot in an instance of* TbC*. In any terminal configuration of this instance, at least one process of $\mathcal{C}$ verifies $P\_root(u)$. In particular, every process u that has the smallest d_u value in $\mathcal{C}$ verifies $P\_root(u)$.*

7.1 Spanning Forest and Non-Rooted Components Detection

Given an input set of processes *rootSet*, Algorithm Forest is the instantiation of TbC 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), whose tree roots are processes of *rootSet*. Each process of

Algorithm 2: Parameters for any process u in Algorithm Forest – Versions 1 and 2

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 Magma:

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

Predicates:

- $P\_neighActive(u) \equiv false$
- In **Version 1**, $P\_rootActive(u) \equiv P\_rootValid(u)$
  In **Version 2**, $P\_rootActive(u) \equiv false$
- $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C)$

$rootSet$ is required to be a tree root in Version 1 of Algorithm Forest, but not in Version 2. Moreover, in any component containing no process of $rootSet$, the processes eventually detect the absence of root by finally taking the status I (Isolated).

Correctness of Forest. In a terminal 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, $st_u = C$, $par_u \in \Gamma(u)$, $d_u \geq d_{par_u} + 1$, u belongs to a tree rooted at some process of $rootSet$ – its neighbor par_u is its parent in the tree. In Version 1 of the algorithm, $u \notin rootSet$.
3. There is no process of $rootSet$ in V_u and $st_u = I$, *i.e.*, u is isolated.

Move Complexity of Forest. Rule $\mathbf{R_U}$ is executed at most once by each process u in any V_u-segment. Hence, the total number of moves (and steps) during any execution is bounded by $5 \cdot (n_{\texttt{maxCC}} + 1) \cdot n$, by Theorem 2 (page 22).

7.2 Leader Election

Assuming the network is identified, Algorithm LE is the instantiation of TbC with the parameters given in Algorithm 3. In each connected component, Algorithm LE elects a process ℓ (*i.e.*, $P\_leader(\ell)$ holds) and builds a tree (of arbitrary topology) rooted at ℓ that spans the whole connected component.

Algorithm 3: Parameters for any process u in Algorithm LE

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

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

Predicates:
- $P\_neighActive(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C \wedge d_v.id < d_u.id)$
- $P\_rootActive(u) \equiv false$
- $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C)$
- $P\_leader(u) \equiv P\_root(u)$

The variable d_u of a process u has two fields. The first one, id, eventually contains the identifier of the leader in V_u. The second one, h, contains an upper bound on the distance to the leader in the built tree rooted at ℓ.

Correctness of LE. As *canBeRoot* is true for all processes, in a terminal configuration of LE no process verifies Property 3 (*i.e.* no process has the status I).

Observation 10 *In a terminal configuration of* LE*, each process u satisfies one of the following conditions: (1) $P\_root(u)$, or (2) $st_u = C$, $par_u \in \Gamma(u)$, $d_u = (d_{par_u}.id, -)$, and $d_u.h \geq d_{par_u}.h + 1$.*

In a terminal configuration of Algorithm LE, each process u satisfies $d_u = (pname_\ell, -)$ where ℓ is the single process in V_u verifying $P\_root(\ell)$.

Move Complexity of LE. During a $\mathcal{C}$-segment, a process can only execute $\mathbf{R_U}$ to improve its ID. Let u by any process. At the beginning of a segment, at most $n_{\texttt{maxCC}} - 1$ distinct IDs are stored in the distance variables of processes in $V_u \setminus \{u\}$. In the worst case, u can successively adopt each of them along the segment. Hence, the total number of moves (and steps) during any execution is bounded by $(n_{\texttt{maxCC}}+3)\cdot(n_{\texttt{maxCC}}+1)\cdot n$ (Theorem 2, page 22), *i.e.*, $O({n_{\texttt{maxCC}}}^2 \cdot n)$.

7.3 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 TbC 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 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 the built tree is a shortest path from u to r in V_r.

Algorithm 4: Parameters for any process u in Algorithm RSP

Inputs:

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

Ordered Magma:

The same as for Algorithm Forest (Algorithm 2)

Predicate:

- $P\_neighActive(u) \equiv P\_neighValid(u)$
- $P\_rootActive(u) \equiv P\_rootValid(u)$
- $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C)$

Correctness of RSP. By definition, r is the unique process satisfying $canBeRoot$. So, only r can satisfy P_root. By Observation 9, $P\_root(r)$ holds in any terminal configuration of RSP.

Observation 11 *In a terminal configuration of Algorithm* RSP*, each process u satisfies one of the following three conditions: (1) $u = r$ and $P\_root(r)$ holds; (2) $u \in V_r \setminus \{r\}$, $st_u = C$, $par_u \in \Gamma(u)$, and $d_u = d_{par_u} + \omega_u(par_u)$; or (3) $u \notin V_r$ and $st_u = I$.*

In a terminal configuration of Algorithm RSP, each process u satisfies $u \notin V_r$, or $d_u = d(u, r)$.

Move Complexity of RSP. All edge weights are strictly positive and $\oplus$ is the addition operator, so the total number of moves (and steps) during any execution is bounded by $(\mathtt{W}_{\max} \cdot (n_{\mathtt{maxCC}} - 1)^2 + 5) \cdot (n_{\mathtt{maxCC}} + 1) \cdot n$ (Theorem 3, page 25), *i.e.*, $O({n_{\mathtt{maxCC}}}^3 \cdot n \cdot \mathtt{W}_{\max})$.

Construction of BFS tree rooted at r. If all edge weights have the same value (for instance, 1) then Algorithm RSP builds a BFS tree rooted at r in V_r. In any other component, the processes take the status I (Isolated). As edges have the same weight, the total number of moves (and steps) during any execution is bounded by $((n_{\texttt{maxCC}} - 1)^2 + 5) \cdot (n_{\texttt{maxCC}} + 1) \cdot n$ (Corollary 8, page 25), *i.e.*, $O({n_{\texttt{maxCC}}}^3 \cdot n)$.

Bounded Memory Space version of RSP. Corollary 3 (page 19) describes the instantiation of TbC requiring only bounded memory space that is similar to RSP (*i.e.*, a terminal configuration of this instantiation is a terminal configuration of RSP). The set $distSetFinite$ can be any finite set containing every distance d that may be assumed in a terminal configuration of RSP. Therefore, each process only needs to know an upper bound BL on the maximum weighted distance to r. This corresponds to setting $distSetFinite = \{d \in \mathbb{N} \mid d \leq \texttt{BL}\}$. This could also be done by providing an upper bound B on the network size and an upper bound BW on the edge weights, and setting $\texttt{BL} = B \cdot \texttt{BW}$. Finally, the move complexity of the Bounded Memory Space version of RSP remains the same as the one of the initial version of RSP, *i.e.*, $O({n_{\texttt{maxCC}}}^3 \cdot n \cdot \texttt{W}_{\max})$.

7.4 Leader Election of the process with the smallest identifier and Shortest-Path-Tree construction requiring bounded memory space

Assuming the network is identified, Algorithm LEM_SP_BD is the instantiation of TbC with the parameters given in Algorithm 5. The variable d has two fields. The first one, id, eventually contains the identifier of the elected process. The second one, h, contains the weighted distance to the elected process.

In each connected component, Algorithm LEM_SP_BD elects (in a self-stabilizing manner) the process ℓ (*i.e.*, $P\_leader(\ell)$ holds) of smallest identifier and builds a shortest-path tree rooted at ℓ.

The memory space required by Algorithm LEM_SP_BD on each process is bounded by $O(\log(B \cdot \texttt{BW}))$ bits, where B is an upper bound on the maximum value of a process identifier and BW is an upper bound on the edge weights (an upper bound on the maximum weighted distance could also be used; see the discussion on that matter in Section 7.3). However, each process needs to know both B and BW.

Correctness of LEM_SP_BD. As *canBeRoot* is true for all processes, in a terminal configuration of LEM_SP_BD, no process has the status I. According to Corollary 3 (page 19), we have the following observation.

Observation 12 *In a terminal configuration of Algorithm* LEM_SP_BD*, each process u satisfies one of the following conditions: (1) $P\_root(u)$, or (2) $st_u = C$, $par_u \in \Gamma(u)$, $d_u = (d_{par_u}.id, d_{par_u}.h + \omega_u(par_u))$.*

Algorithm 5: Parameters for any process u in Algorithm LEM_SP_BD

Inputs:

- $canBeRoot_u$ is true for any process
- $pname_u$ is the identifier of u (*n.b.*, $pname_u \in \mathbb{N}$)
- $\omega_u(v) = (\bot, weight_u(v))$ for every $v \in \Gamma(u)$ with $weight_v(u) = weight_u(v) \in \mathbb{N}^*$ being the weight of the edge $\{u, v\}$
- $distSetFinite = \{(id, h) \in \mathbb{N}^2 \mid id \leq B \text{ and } h \leq B \cdot \texttt{BW}\}$
  B is an upper bound on the maximum identifier and
  BW is an upper bound on the edge weights

Ordered Magma:

The same as for Algorithm LE (Algorithm 3)

Predicates:

- $P\_neighActive(u) \equiv P\_neighValid(u) \wedge P\_toBeC(u)$
- $P\_rootActive(u) \equiv P\_rootValid(u)$
- $P\_toBeC(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C) \wedge distNeigh(u) \in distSetFinite$
- $P\_leader(u) \equiv P\_root(u)$

In a terminal configuration of Algorithm LEM_SP_BD, for every process u of $\mathcal{C}$, $d_u = (pname_\ell, d(u, \ell))$ with ℓ being the process having the smallest identifier in $\mathcal{C}$.

Move Complexity of LEM_SP_BD. $SI_{\mathcal{S},v}$ is the set of d values obtained after executing an action belonging to the maximal causal chains rooted at v in the segment $\mathcal{S}$; see definition 13.
$SI_{\mathcal{S},v} \subseteq \{ds_{\mathcal{S},v} \oplus (i \cdot (\bot, 1)) \mid 1 \leq i \leq \mathtt{W}_{\max} \cdot (n_{\mathtt{maxCC}} - 1)\}$, $ds_{\mathcal{S},v}$ being the initiating value common to all maximal causal chains of $\mathcal{S}$ rooted at v. So, $|SI_{\mathcal{S},v}| \leq \mathtt{W}_{\max} \cdot (n_{\mathtt{maxCC}} - 1)$. According to Corollary 6 (page 24), the total number of moves (and steps) during any execution is bounded by $(\mathtt{W}_{\max} \cdot (n_{\mathtt{maxCC}} - 1)^2 + 5) \cdot (n_{\mathtt{maxCC}} + 1) \cdot n$, *i.e.*, $O({n_{\mathtt{maxCC}}}^3 \cdot n \cdot \mathtt{W}_{\max})$.

Construction of a BFS tree rooted at the process having the smallest identifier and requiring bounded memory space. If all edge weights have the same value (for instance, 1) then in each connected component $\mathcal{C}$, Algorithm LEM_SP_BD builds a BFS tree rooted at the process having the smallest identifier of $\mathcal{C}$. As edges have the same weight, the total number of moves (and steps) during any execution is bounded by $((n_{\mathtt{maxCC}} - 1)^2 + 5) \cdot (n_{\mathtt{maxCC}} + 1) \cdot n$ (Corollary 8, page 25), *i.e.*, $O({n_{\mathtt{maxCC}}}^3 \cdot n)$.

Version of LEM_SP_BD *without any knowledge about the identifiers and the edge weights.* Corollary 3 (page 19) presents the instantiation of TbC not requiring any network knowledge and similar to LEM_SP_BD (*i.e.* a terminal

configuration of this instantiation is a terminal configuration of LEM_SP_BD). The memory space required by this instance on each process is unbounded though. Indeed, given any bound B on the second component h of a distance, such a value B for h could be initially present at some process u and another process v might connect to u and obtain a second component larger than the bound B. The move complexity of this instance is also $O({n_{\texttt{maxCC}}}^3 \cdot n \cdot \mathtt{W}_{\max})$.

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

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

Here, the *weight of the arc* (u, v) is $\alpha_u(v)$, the local label of u in $\Gamma(v)$. Let $\mathcal{P} = u_k, u_{k-1}, \ldots u_0 = r$ be a (directed) path from process u_k to the root r. We define the *weight of* $\mathcal{P}$ as the sequence $0, \alpha_1(u_0), \alpha_2(u_1), \ldots, \alpha_k(u_{k-1})$. 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).

Algorithm 6: Parameters for any process u in Algorithm RDFS

Inputs:

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

Ordered Magma:

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

Predicate:

The same as for Algorithm RSP (Algorithm 4)

Correctness of RDFS. 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. $u \neq r$, $u \in V_r$, $st_u = C$, $par_u \in \Gamma(u)$, *and* $d_u = d_{lex}(u, r) = d_{par_u}.\omega_u(par_u)$; *or*
3. $u \notin V_r$ *and* $st_u = I$.

Let T be a tree rooted at r that spans V_r. Following the result of [12], 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.

Move Complexity of RDFS. For this instance, we cannot apply Corollary 6 (page 24) to obtain a polynomial move complexity. However, by Lemma 14 we have a rough estimation of the move complexity, *i.e.*, at most $(n_{\texttt{maxCC}} - 1)!$ moves. We outline that this estimation is coarse-grained, and so can be further refined.

Bounded Memory Space version of RDFS. Corollary 3 (page 19) describes the instantiation of TbC requiring only bounded memory space that is similar to RDFS (*i.e.*, a terminal configuration of this instantiation is a terminal configuration of RDFS). The set $distSetFinite$ can be any finite set containing every distance d that may be assumed in a terminal configuration of RDFS. Therefore, each process only needs to know an upper bound B on the network size and an upper bound BD on Δ (the maximum degree of G). This corresponds to setting $distSetFinite = \{0, \ldots, \texttt{BD}\}^{\leq B}$, the set of words of length at most B over the alphabet $\{0, \ldots, \texttt{BD}\}$.

7.6 Optimum-bandwidth-path (spanning) tree

Assume the existence of a unique root r. Algorithm RBW is the instantiation of TbC with the parameters given in Algorithm 7. The variable d on v contains the *multiset* of the edge bandwidths in the path from v to r. Note that storing only the bottleneck bandwidth or the set of edge bandwidths (even combined with the distance to r) would not satisfy the constraints on $\oplus$ and $\prec$ stated in Fig. 1.

Algorithm RBW computes (in a self-stabilizing manner) a spanning tree rooted at r in V_r. The path from u to r in the spanning tree is one that maximizes the bandwidth. Moreover, in any other component, processes eventually detect the absence of r by taking the status I (Isolated).

The *bandwidth of a path* is the minimum of its edge bandwidths. The *bandwidth capability* between the processes u and v, denoted by $bwc(u, v)$, is the maximum bandwidth of a path from u to v. An *optimum bandwidth path* from u to v is a path whose bandwidth is $bwc(u, v)$. An *optimum-bandwidth-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 the built tree is an optimum-bandwidth-path from u to r in V_r.

Algorithm 7: Parameters for any process u in Algorithm RBW

Inputs:

- $canBeRoot_u$ is false for any process except for $u = r$
- $pname_u$ is $\perp$
- $\omega_u(v) = \{bw_u(v)\}$, for every $v \in \Gamma(u)$ with $bw_u(v) \in \mathbb{N}^*$ is the bandwidth of the directed link from u to v

Ordered Magma:

- $DistSet$ = the set of the finite multisets of elements in $\mathbb{N}^*$
- $d_1 \oplus d_2 = d_1 \uplus d_2$, where $\uplus$ is the sum (disjoint union) on multisets
- Inductive definition of the total order $\prec$ on $DistSet$:
  $d_1 \prec d_2 \equiv (d_1 = \emptyset \wedge d_2 \neq \emptyset) \vee \big[d_1 \neq \emptyset \wedge d_2 \neq \emptyset \wedge$
  $[\min d_1 > \min d_2 \vee (\min d_1 = \min d_2 \wedge d_1 \setminus \{\min d_1\} \prec d_2 \setminus \{\min d_2\})]\big]$
  ($s \setminus \{x\}$ is the multiset s in which *one* occurrence of x has been removed)
- $distRoot(u) = \emptyset$, the empty multiset

Predicate:

The same as for Algorithm RSP (Algorithm 4)

Correctness of RBW. First note that the ordered magma and the weight assignment satisfy the constraints on $\oplus$ and $\prec$ stated in Fig. 1. By definition, r is the unique process satisfying $canBeRoot$. So, only r can satisfy P_root. By Observation 9, $P\_root(r)$ holds in any terminal configuration of RBW.

Observation 14 *In a terminal configuration of Algorithm* RBW*, each process u satisfies one of the following three conditions:*

1. $u = r$ *and* $P\_root(r)$ *holds;*
2. $u \in V_r \setminus \{r\}$, $par_u \in \Gamma(u)$, $d_u = d_{par_u} \uplus \{bw_u(v)\}$, *and* $st_u = C$*; or*
3. $u \notin V_r$ *and* $st_u = I$.

In a terminal configuration of Algorithm RBW, each process u satisfies $u \notin V_r$, or d_u is the multiset of the edge bandwidths of a path such that $\min d_u = bwc(u, r)$.

Move Complexity of RBW. $SI_{\mathcal{S},v}$ is the set of distance values obtained after executing an action belonging to the maximal causal chains rooted at v in the segment $\mathcal{S}$; see Definition 13. Let k be the number of different edge bandwidths in the network. The size of $SI_{\mathcal{S},v}$ is at most the number of multisets of at most $n_{\texttt{maxCC}} - 1$ elements from the set of the k possible edge bandwidths, such a multiset being added (by successive disjoint unions of singletons) to the initiating multiset $ds_{\mathcal{S},v}$. So the size of $SI_{\mathcal{S},v}$ is bounded by $\binom{k+n_{\texttt{maxCC}}-1}{n_{\texttt{maxCC}}-1} = \binom{k+n_{\texttt{maxCC}}-1}{k}$. According to Corollary 6 (page 24), the total number of moves during any execution, is bounded by $(\binom{k+n_{\texttt{maxCC}}-1}{k}(n_{\texttt{maxCC}} - 1) + 5)(n_{\texttt{maxCC}} + 1)n$. In practice, the number k of different bandwidth values may be small (one per

technology for example), or the bandwidth values may be grouped in a small number of ranges. When k is constant, the total number of moves during any execution becomes polynomial, in $O({n_{\texttt{maxCC}}}^{k+2} \cdot n)$.

Bounded Memory Space version of RBW. Corollary 3 (page 19) describes the instantiation of TbC requiring only bounded memory space similar to RBW (*i.e.*, a terminal configuration of this instantiation is a terminal configuration of RBW). The set $distSetFinite$ can be any finite set containing every distance d that may be assumed in a terminal configuration of RBW, for example the set of all multisets of at most $n_{\texttt{maxCC}}-1$ elements from the set of the k possible edge bandwidths. Therefore, each process only needs to know an upper bound B on the network size and an upper bound BW on the edge bandwidths. Even better than the knowledge of BW would be the complete knowledge of the k possible edge bandwidths. In this case, and when k is constant, the memory requirement for the distance d value would be logarithmic in $n_{\texttt{maxCC}}$.

8 Conclusion

We proposed a general scheme, called Algorithm TbC, to compute tree-based data structures on arbitrary (not necessarily connected) bidirectional networks.

Algorithm TbC is self-stabilizing and silent. It is written in the locally shared memory model with composite atomicity. We have proven its correctness under the distributed unfair daemon hypothesis, the weakest scheduling assumption of the model. We have also shown that its stabilization time is at most $4n_{\texttt{maxCC}}$ rounds, where $n_{\texttt{maxCC}}$ is the maximum number of processes in a connected component.

We illustrated the versatility of our approach by proposing several instantiations of TbC that solve classical problems in various settings. For example, we can instantiate TbC to solve leader election and/or spanning tree or forest constructions in identified or semi-anonymous (*e.g.*, rooted) networks. These spanning structures may be of different types, *e.g.*, arbitrary, BFS, DFS, shortest-path, ... Note also that, whenever the network is not connected, TbC also achieves the non-rooted components detection.

In most of the cases, we exhibited polynomial upper bounds on the stabilization time in steps and process moves of the considered instantiations. Finally, in many cases, instantiations can be easily modified to handle bounded local memories, without any overhead.

References

1. Altisen, K., Cournier, A., Devismes, S., Durand, A., Petit, F.: Self-stabilizing leader election in polynomial steps. Information and Computation **254**, 330–366 (2017). DOI 10.1016/j.ic.2016.09.002. URL https://doi.org/10.1016/j.ic.2016.09.002

2. Altisen, K., Devismes, S., Dubois, S., Petit, F.: Introduction to Distributed Self-Stabilizing Algorithms. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2019). DOI 10.2200/S00908ED1V01Y201903DCT015. URL https://doi.org/10.2200/S00908ED1V01Y201903DCT015
3. Arora, A., Gouda, M., Herman, T.: Composite routing protocols. In: the 2nd IEEE Symposium on Parallel and Distributed Processing (SPDP'90), pp. 70–78 (1990)
4. Beauquier, J., Gradinariu, M., Johnen, C.: Cross-over composition - enforcement of fairness under unfair adversary. In: 5th International Workshop on Self-Stabilizing Systems, (WSS 2001), Springer LNCS 2194, pp. 19–34 (2001)
5. Blin, L., Cournier, A., Villain, V.: An improved snap-stabilizing PIF algorithm. In: S. Huang, T. Herman (eds.) Self-Stabilizing Systems, 6th International Symposium, SSS 2003, *Lecture Notes in Computer Science*, vol. 2704, pp. 199–214. Springer, San Francisco, CA, USA (2003)
6. Blin, L., Fraigniaud, P., Patt-Shamir, B.: 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, pp. 18–32 (2014)
7. Blin, L., Potop-Butucaru, M., Rovedakis, S., Tixeuil, S.: 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, pp. 50–64 (2010)
8. Boldi, P., Vigna, S.: Universal dynamic synchronous self–stabilization. Distributed Computing **15**(3), 137–153 (2002)
9. Carrier, F., Datta, A.K., Devismes, S., Larmore, L.L., Rivierre, Y.: Self-stabilizing (f, g)-alliances with safe convergence. J. Parallel Distrib. Comput. **81-82**, 11–23 (2015). DOI 10.1016/j.jpdc.2015.02.001. URL http://dx.doi.org/10.1016/j.jpdc.2015.02.001
10. Chang, E.J.H.: Echo Algorithms: Depth Parallel Operations on General Graphs. IEEE Trans. Software Eng. **8**(4), 391–401 (1982)
11. Cobb, J.A., Huang, C.T.: Stabilization of Maximal-Metric Routing without Knowledge of Network Size. In: 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 306–311 (2009). DOI 10.1109/PDCAT.2009.75
12. Collin, Z., Dolev, S.: Self-stabilizing depth-first search. Inf. Process. Lett. **49**(6), 297–301 (1994)
13. Cournier, A.: A new polynomial silent stabilizing spanning-tree construction algorithm. In: International Colloquium on Structural Information and Communication Complexity, pp. 141–153. Springer (2009)
14. Cournier, A., Datta, A.K., Devismes, S., Petit, F., Villain, V.: The expressive power of snap-stabilization. Theor. Comput. Sci. **626**, 40–66 (2016). DOI 10.1016/j.tcs.2016.01.036. URL https://doi.org/10.1016/j.tcs.2016.01.036
15. Cournier, A., Devismes, S., Petit, F., Villain, V.: Snap-stabilizing depth-first search on arbitrary networks. The Computer Journal **49**(3), 268–280 (2006)
16. Cournier, A., Devismes, S., Villain, V.: A snap-stabilizing dfs with a lower space requirement. In: Symposium on Self-Stabilizing Systems, pp. 33–47. Springer (2005)
17. Cournier, A., Devismes, S., Villain, V.: Light enabling snap-stabilization of fundamental protocols. TAAS **4**(1), 6:1–6:27 (2009). DOI 10.1145/1462187.1462193. URL http://doi.acm.org/10.1145/1462187.1462193
18. Cournier, A., Devismes, S., Villain, V.: Light enabling snap-stabilization of fundamental protocols. ACM Transactions on Autonomous and Adaptive Systems **4**(1) (2009)
19. Cournier, A., Rovedakis, S., Villain, V.: 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, pp. 159–174 (2011)
20. Cournier, Alain: A lower bound for the $Max + 1$ algorithm. https://home.mis.u-picardie.fr/~cournier/MaxPlusUn.pdf (2009). Online; accessed 11 February 2009
21. Datta, A.K., Devismes, S., Heurtefeux, K., Larmore, L.L., Rivierre, Y.: Competitive self-stabilizing k-clustering. Theor. Comput. Sci. **626**, 110–133 (2016). DOI 10.1016/j.tcs.2016.02.010. URL https://doi.org/10.1016/j.tcs.2016.02.010
22. Datta, A.K., Gurumurthy, S., Petit, F., Villain, V.: Self-stabilizing network orientation algorithms in arbitrary rooted networks. Stud. Inform. Univ. **1**(1), 1–22 (2001)
23. Datta, A.K., Larmore, L.L., Devismes, S., Heurtefeux, K., Rivierre, Y.: Self-stabilizing small k-dominating sets. IJNC **3**(1), 116–136 (2013)

24. Datta, A.K., Larmore, L.L., Vemula, P.: An o(n)-time self-stabilizing leader election algorithm. jpdc **71**(11), 1532–1544 (2011)
25. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing leader election in optimal space under an arbitrary scheduler. Theoretical Computer Science **412**(40), 5541–5561 (2011)
26. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators revisited. Journal of Aerospace Computing, Information, and Communication (JACIC) **3**(10), 498–514 (2006). DOI 10.2514/1.19848
27. Devismes, S., Ilcinkas, D., Johnen, C.: Self-stabilizing disconnected components detection and rooted shortest-path tree maintenance in polynomial steps. In: P. Fatourou, E. Jiménez, F. Pedone (eds.) 20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain, *LIPIcs*, vol. 70, pp. 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
28. Devismes, S., Ilcinkas, D., Johnen, C.: Silent self-stabilizing scheme for spanning-tree-like constructions. In: R.C. Hansdah, D. Krishnaswamy, N. Vaidya (eds.) Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019, pp. 158–167. ACM (2019)
29. Devismes, S., Johnen, C.: Silent self-stabilizing {BFS} tree algorithms revisited. Journal of Parallel and Distributed Computing **97**, 11 – 23 (2016). DOI http://dx.doi.org/10.1016/j.jpdc.2016.06.003. URL `http://www.sciencedirect.com/science/article/pii/S0743731516300685`
30. Dijkstra, E.W.: Self-stabilizing Systems in Spite of Distributed Control. Commun. ACM **17**(11), 643–644 (1974)
31. Dolev, S.: Self-stabilization. MIT Press (2000)
32. Dolev, S., Gouda, M.G., Schneider, M.: Memory requirements for silent stabilization. Acta Informatica **36**(6), 447–462 (1999)
33. Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators. Distributed Computing **14**(3), 147–162 (2001). DOI 10.1007/PL00008934. URL `https://doi.org/10.1007/PL00008934`
34. Glacet, C., Hanusse, N., Ilcinkas, D., Johnen, C.: 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, pp. 120–134 (2014)
35. Glacet, C., Hanusse, N., Ilcinkas, D., Johnen, C.: Disconnected components detection and rooted shortest-path tree maintenance in networks. Journal of Parallel and Distributed Computing (2019). DOI https://doi.org/10.1016/j.jpdc.2019.05.006. URL `http://www.sciencedirect.com/science/article/pii/S0743731519303934`
36. Godard, E.: Snap-Stabilizing Tasks in Anonymous Networks. In: Stabilization, Safety, and Security of Distributed Systems (SSS'16), Lecture Notes in Computer Science, pp. 170–184. Springer, Cham (2016)
37. Gouda, M.G., Herman, T.: Adaptive programming. IEEE Trans. Software Eng. **17**(9), 911–921 (1991)
38. Gärtner, F.C.: A survey of self-stabilizing spanning-tree construction algorithms. Tech. rep., Swiss Federal Institute of Technolog (EPFL) (2003)
39. Huang, S.T., Chen, N.S.: A self-stabilizing algorithm for constructing breadth-first trees. Information Processing Letters **41**(2), 109–117 (1992)
40. Ilcinkas, D., Johnen, C., Laplace, R.: STlC meta-algorithm on JBotSim, SWHID: ⟨swh:1:rev:65238c18e332117de59682903f828be18ad6a91f;origin=https://gitlab.com/remikey/jbotsim-stlc.git⟩. URL `https://archive.softwareheritage.org/{SWHID}`
41. Katz, S., Perry, K.J.: Self-stabilizing extensions for message-passing systems. Distributed Computing **7**(1), 17–26 (1993)
42. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Distributed Computing **22**(4), 215–233 (2010). DOI 10.1007/s00446-010-0095-3. URL `https://doi.org/10.1007/s00446-010-0095-3`
43. Kosowski, A., Kuszner, L.: 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, pp. 75–82 (2005)
44. Leon-Garcia, A., Widjaja, I.: Communication Networks, 2 edn. McGraw-Hill, Inc., New York, NY, USA (2004)

45. Segall, A.: Distributed Network Protocols. IEEE Transactions on Information Theory **29**(1), 23–34 (1983)
46. Sloman, M., Kramer, J.: Distributed systems and computer networks. Prentice Hall (1987)
47. Tel, G.: Introduction to distributed algorithms. Cambridge University Press, Cambridge, UK (Second edition 2001)