Self-Stabilizing Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Polynomial Steps*

Stéphane Devismes[1]   David Ilcinkas[2]   Colette Johnen[2]

[1] *Université Grenoble Alpes, Grenoble, France*
[2] *Univ. Bordeaux & CNRS, LaBRI, UMR 5800, F-33400 Talence, France*

We deal with the problem of maintaining a shortest-path tree rooted at some process r in a network that may be disconnected after topological changes. The goal is then to maintain a shortest-path tree rooted at r in its connected component, V_r, and make all processes of other components detecting that r is not part of their connected component. We propose, in the composite atomicity model, a silent self-stabilizing algorithm for this problem working in semi-anonymous networks, where edges have strictly positive weights. This algorithm does not require any *a priori* knowledge about global parameters of the network. We prove its correctness assuming the distributed unfair daemon, the most general daemon. Its stabilization time in rounds is at most $3n_{\mathrm{maxCC}} + D$, where n_{maxCC} is the maximum number of non-root processes in a connected component and D is the hop-diameter of V_r. Furthermore, if we additionally assume that edge weights are positive integers, then it stabilizes in a polynomial number of steps: namely, we exhibit a bound in $O(\mathrm{W}_{\max} n_{\mathrm{maxCC}}{}^3 n)$, where $\mathrm{W}_{\max}$ is the maximum weight of an edge and n is the number of processes.

Keywords: distributed algorithm, self-stabilization, routing algorithm, shortest path, disconnected network, shortest-path tree

1   Introduction

Given a connected undirected edge-weighted graph G, a *shortest-path (spanning) tree rooted at node r* is a spanning tree T of G, such that for every node u, the unique path from u to r in T is a shortest path from u to r in G. This data structure finds applications in the networking area (*n.b.*, in this context, nodes actually represent processes), since many distance-vector routing protocols, like *RIP* (*Routing Information Protocol*) and *BGP* (*Border Gateway Protocol*), are based on the construction of shortest-path trees. Indeed, such algorithms implicitly builds a shortest-path tree rooted at each destination.

From time to time, the network may be split into several connected components due to the network dynamics. In this case, routing to process r correctly operates only for the processes of its connected component, V_r. Consequently, in other connected components, information to reach r should be removed to gain space in routing tables, and to discard messages destined to r (which are unable to reach r anyway) and thus save bandwidth. The goal is then to make the network converging to a configuration where every process of V_r knows a shortest path to r and every other process detects that r is not in its own connected component. We call this problem the *Disconnected Components Detection and rooted Shortest-Path tree Maintenance* (DCDSPM) problem. Notice that a solution to this problem allows to prevent the well-known *count-to-infinity* problem [LGW04], where the distances to some unreachable process keep growing in routing tables because no process is able to detect the issue.

When topological changes are infrequent, they can be considered as transient faults [Tel01] and self-stabilization [Dij74] — a versatile technique to withstand *any* finite number of transient faults in a distributed system — becomes an attractive approach. A self-stabilizing algorithm is able to recover without external (*e.g.*, human) intervention a correct behavior in finite time, regardless of the *arbitrary* initial configuration of the system, and therefore, also after the occurrence of transient faults, provided that these faults do not alter the code of the processes.

A particular class of self-stabilizing algorithms is that of silent algorithms. A self-stabilizing algorithm is *silent* [DGS99] if it converges to a global state where the values of communication registers used by the algorithm remain fixed. Silent (self-stabilizing) algorithms are usually proposed to build distributed data structures, and so are well-suited for the problem considered here. As quoted in [DGS99], the silent property usually implies more simplicity in the algorithm design, moreover a silent algorithm may utilize less communication operations and communication bandwidth.

For the sake of simplicity, we consider here a single destination process r, called the *root*. However, the solution we will propose can be generalized to work with any number of destinations, provided that destinations can be distinguished. In this context, we do not require the network to be fully identified. Rather, r should be distinguished among other processes, and all non-root processes are supposed to be identical: we consider semi-anonymous networks.

In this paper, we propose a silent self-stabilizing algorithm, called Algorithm RSP, for the DCDSPM problem with a single destination process in semi-anonymous networks. Algorithm RSP does not require any *a priori* knowledge of processes about global parameters of the network, such as its size or its diameter. Algorithm RSP is written in the locally shared memory model with composite atomicity introduced by Dijkstra [Dij74], which is the most commonly used model in self-stabilization. In this model, executions proceed in (atomic) steps, and a self-stabilizing algorithm is silent if and only if all its executions are finite. Moreover, the asynchrony of the system is captured by the notion of *daemon*. The weakest (*i.e.*, the most general) daemon is the *distributed unfair daemon*. Hence, solutions stabilizing under such an assumption are highly desirable, because they work under any other daemon assumption. Interestingly, self-stabilizing algorithms designed under this assumption are easier to compose (composition techniques are widely used to design and prove complex self-stabilizing algorithms). Moreover, time complexity (the *stabilization time*, mainly) can be bounded in terms of steps only if the algorithm works under an unfair daemon. Otherwise (*e.g.*, under a weakly fair daemon), time complexity can only be evaluated in terms of rounds, which capture the execution time according to the slowest process. There are many self-stabilizing algorithms proven under the distributed unfair daemon, *e.g.*, [ACD$^+$17, CDD$^+$15, DLV11a, DLV11b, GHIJ14]. However, analyses of the stabilization time in steps is rather unusual and this may be an important issue. Indeed, this complexity captures the amount of computations an algorithm needs to recover a correct be-

havior. Now, recently, several self-stabilizing algorithms, which work under a distributed unfair daemon, have been shown to have an exponential stabilization time in steps in the worst case. In [ACD$^+$17], silent leader election algorithms from [DLV11a, DLV11b] are shown to be exponential in steps in the worst case. In [DJ16], the Breadth-First Search (BFS) algorithm of Huang and Chen [HC92] is also shown to be exponential in steps. Finally, in [Ga16] authors show that the first silent self-stabilizing algorithm for the DCDSPM problem (still assuming a single destination) they proposed in [GHIJ14] is also exponential in steps.

1.1 Contribution

Algorithm RSP proposed here is proven assuming the distributed unfair daemon. We also study its stabilization time in rounds. We establish a bound of at most $3n_{\mathrm{maxCC}} + D$ rounds, where n_{maxCC} is the maximum number of non-root processes in a connected component and D is the hop-diameter of V_r (defined as the maximum over all pairs $\{u, v\}$ of nodes in V_r of the minimum number of edges in a shortest path from u to v).

   Furthermore, RSP is the first silent self-stabilizing algorithm for the DCDSPM problem which, assuming that the edge weights are positive integers, achieves a polynomial stabilization time in steps. Namely, in this case, the stabilization time of RSP is at most $(\mathtt{W}_{\max}n_{\mathrm{maxCC}}{}^3 + (3 - \mathtt{W}_{\max})n_{\mathrm{maxCC}} + 3)(n-1)$, where $\mathtt{W}_{\max}$ is the maximum weight of an edge and n is the number of processes. (*N.b.*, this stabilization time is less than or equal to $\mathtt{W}_{\max}n^4$, for all $n \geq 3$.)

   Finally, notice that when all weights are equal to one, the DCDSPM problem reduces to a BFS tree maintenance and the step complexity becomes at most $(n_{\mathrm{maxCC}}{}^3 + 2n_{\mathrm{maxCC}} + 3)(n-1)$, which is less than or equal to n^4 for all $n \geq 2$.

1.2 Related Work

To the best of our knowledge, only one self-stabilizing algorithm for the DCDSPM problem has been previously proposed in the literature [GHIJ14]. This algorithm is silent and works under the distributed unfair daemon, but, as previously mentioned, it is exponential in steps. However, it has a slightly better stabilization time in rounds, precisely at most $2(n_{\mathrm{maxCC}} + 1) + D$ rounds[i].

   There are several shortest-path spanning tree algorithms in the literature that do not consider the problem of disconnected components detection. The oldest distributed algorithms are inspired by the Bellman-Ford algorithm [Bel58, FJ56]. Self-stabilizing shortest-path spanning tree algorithms have then been proposed in [CS94, HL02], but these two algorithms are proven assuming a central daemon, which only allows sequential executions. However, in [Hua05b], Tetz Huang proves that these algorithms actually work assuming the distributed unfair daemon. Nevertheless, no upper bounds on the stabilization time (in rounds or steps) are given. More recently, Cobb and Huang [CH09] proposed an algorithm constructing shortest-path trees based on any maximizable routing metrics. This algorithm does not require a priori knowledge about the network but it is proven only for the central weakly-fair daemon. It runs in a linear number of rounds and no analysis is given on the number of steps.

   Self-stabilizing shortest-path spanning tree algorithms are also given in [AGH90, CG02, JT03]. These algorithms additionally ensure the loop-free property in the sense that they guarantee that a spanning tree structure is always preserved while edge costs change dynamically. However, none of these papers consider the unfair daemon, and consequently their step complexity cannot be analyzed.

---

[i]In fact, [GHIJ14] announced $2n + D$ rounds, but it is easy to see that this complexity can be reduced to $2(n_{\mathrm{maxCC}} + 1) + D$.

Whenever all edges have weight one, shortest-path trees correspond to BFS trees. In [DDL12], the authors introduce the *disjunction* problem as follows. Each process has a constant input bit, 0 or 1. Then, the problem consists for each process in computing an output value equal to the disjunction of all input bits in the network. Moreover, each process with input bit 1 (if any) should be the root of a tree, and each other process should join the tree of the closest input bit 1 process, if any. If there is no process with input bit 1, the execution should terminate and all processes should output 0. The proposed algorithm is silent and self-stabilizing. Hence, if we set the input of a process to 1 if and only if it is the root, then their algorithm solves the DCDSPM problem when all edge-weights are equal to one, since any process which is not in V_r will compute an output 0, instead of 1 for the processes in V_r. The authors show that their algorithm stabilizes in $O(n)$ rounds, but no step complexity analysis is given. Now, as their approach is similar to [DLV11b], it is not difficult to see that their algorithm is also exponential in steps.

Several other self-stabilizing BFS tree algorithms have been proposed, but without considering the problem of disconnected components detection. Chen *et al.* present the first self-stabilizing BFS tree construction in [CYH91] under the central daemon. Huang and Chen present the first self-stabilizing BFS tree construction in [HC92] under the distributed unfair daemon, but recall that this algorithm has been proven to be exponential in steps in [DJ16]. Finally, notice that these two latter algorithms [CYH91, HC92] require that the processes know the exact number of processes in the network.

According to our knowledge, only the following works [CDV09, CRV11] take interest in the computation of the number of steps required by their BFS algorithms. The algorithm in [CDV09] is not silent and has a stabilization time in $O(\Delta n^3)$ steps, where Δ is the maximum degree in the network. The silent algorithm given in [CRV11] has a stabilization time $O(D^2)$ rounds and $O(n^6)$ steps.

Silent self-stabilizing algorithms that construct spanning trees of arbitrary topologies are given in [Cou09, KK05]. The solution proposed in [Cou09] stabilizes in at most $4n$ rounds and $5n^2$ steps, while the algorithm given in [KK05] stabilizes in nD steps (its round complexity is not analyzed).

Several other papers propose self-stabilizing algorithms stabilizing in both a polynomial number of rounds and a polynomial number of steps, *e.g.*, [ACD$^+$17] (for the leader election), [CDPV06, CDV05] (for the DFS token circulation). The silent leader election algorithm proposed in [ACD$^+$17] stabilizes in at most $3n + D$ rounds and $O(n^3)$ steps. DFS token circulations given in [CDPV06, CDV05] execute each wave in $O(n)$ rounds and $O(n^2)$ steps using $O(n \log n)$ 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.

1.3 Roadmap

In the next section, we present the computational model and basic definitions. In Section 3, we describe Algorithm RSP. Its proof of correctness and a complexity analysis in steps are given in Section 4, whereas an analysis of the stabilization time in rounds is proposed in Section 5. Finally, we make concluding remarks in Section 6.

2   Preliminaries

We consider *distributed systems* made of $n \geq 1$ interconnected processes. Each process can directly communicate with a subset of other processes, called its *neighbors*. Communication is assumed to be bidirectional. Hence, the topology of the system can be represented as a simple undirected graph $G = (V, E)$, where V is the set of processes and E the set of edges, representing communication links. Every process v can distinguish its neighbors using a *local labeling* of a given datatype Lbl. All labels of v's

neighbors are stored into the set $\Gamma(v)$. Moreover, we assume that each process v can identify its local label in the set $\Gamma(u)$ of each neighbor u. Such labeling is called *indirect naming* in the literature [SK87]. By an abuse of notation, we use v to designate both the process v itself, and its local labels.

Each edge $\{u, v\}$ has a strictly positive *weight*, denoted by $\omega(u, v)$. This notion naturally extends to paths: the weight of a path in G 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. Of course, $d(u, v) = \infty$ if and only if u and v belong to two distinct connected components of G.

We use the *composite atomicity model of computation* [Dij74, Dol00] in which the processes communicate using a finite number of locally shared registers, called *variables*. Each process can read its own variables and those of its neighbors, but can write only to its own variables. The *state* of a process is defined by the values of its local variables. A *configuration* of the system is a vector consisting of the states of every process.

A *distributed algorithm* consists of one local program per process. We consider semi-uniform algorithms, meaning that all processes except one, the *root* r, execute the same program. In the following, 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_{maxCC} the maximum number of non-root processes in a connected component of G. By definition, $n_{\mathrm{maxCC}} \leq n - 1$.

The *program* of each process consists of a finite set of *rules* of the form $label \; : \; guard \rightarrow action$. *Labels* are only used to identify rules in the reasoning. A *guard* is a Boolean predicate involving the state of the process and that of its neighbors. The *action* part of a rule updates the state of the process. A rule can be executed only if its guard evaluates to *true*; in this case, the rule is said to be *enabled*. A process is said to be enabled if at least one of its rules is enabled. We denote by *Enabled*(γ) 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*(γ) is selected; then every process of $\mathcal{X}$ *atomically* executes one of its enabled rules, leading to a new configuration γ', and so on. The transition from γ to γ' is called a *step*. The possible steps induce a binary relation over $\mathcal{C}$, denoted by $\mapsto$. An *execution* is a maximal sequence of configurations $e = \gamma_0 \gamma_1 \ldots \gamma_i \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration in which no rule is enabled at any process.

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

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

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

- *all executions under S are finite, and*

- *all terminal configurations belong to $\mathcal{L}$.*

We use the notion of *round* [DIM93] to measure the time complexity. The definition of round uses the concept of *neutralization*: a process v is *neutralized* during a step $\gamma_i \mapsto \gamma_{i+1}$, if v is enabled in γ_i but not in configuration γ_{i+1}. Then, the rounds are inductively defined as follows. The first round of an execution $e = \gamma_0, \gamma_1, \cdots$ is the minimal prefix $e' = \gamma_0, \cdots, \gamma_j$, such that every process that is enabled in γ_0 either executes a rule or is neutralized during a step of e'. Let e'' be the suffix $\gamma_j, \gamma_{j+1}, \cdots$ of e. The second round of e is the first round of e'', and so on.

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

3   Algorithm RSP

This section is devoted to the presentation of our algorithm, Algorithm RSP (which stands for *Rooted Shortest-Path*). The code of Algorithm RSP is given in Algorithm 1.

Algorithm 1: Code of RSP

<div style="text-align:center">Macro of RSP for any process u</div>

$$children(u) \quad = \quad \{v \in \Gamma(u) \mid st_u \neq I \wedge st_v \neq I \wedge par_v = u \wedge d_v \geq d_u + \omega(v,u) \wedge (st_v = st_u \vee st_u = EB)\}$$

....

<div style="text-align:center">Code of RSP for the root process r</div>

Constants:
$$\begin{aligned} st_r \quad &= \quad C \\ par_r \quad &= \quad \bot \\ d_r \quad &= \quad 0 \end{aligned}$$

....

<div style="text-align:center">Code of RSP for any process $u \neq r$</div>

Variables:
$$\begin{aligned} st_u \quad &\in \quad \{I, C, EB, EF\} \\ par_u \quad &\in \quad Lbl \\ d_u \quad &\in \quad \mathbb{R}^+ \end{aligned}$$

Predicates:

$abRoot(u) \quad \equiv \quad st_u \neq I \wedge \big[par_u \notin \Gamma(u) \vee st_{par_u} = I \vee d_u < d_{par_u} + \omega(u, par_u) \vee$
$\qquad\qquad\qquad\qquad (st_u \neq st_{par_u} \wedge st_{par_u} \neq EB) \big]$

$P\_reset(u) \quad \equiv \quad st_u = EF \wedge abRoot(u)$

$P\_correction(u) \quad \equiv \quad (\exists v \in \Gamma(u) \mid st_v = C \wedge d_v + \omega(u, v) < d_u)$

Macro:

$computePath(u) \quad : \quad par_u := \mathrm{argmin}_{(v \in \Gamma(u) \wedge st_v = C)}(d_v + \omega(u, v));$
$\qquad\qquad\qquad\qquad d_u := d_{par_u} + \omega(u, par_u);$
$\qquad\qquad\qquad\qquad st_u := C$

Rules

$\mathbf{R_C}(u)$	:	$st_u = C \wedge P\_correction(u)$	$\rightarrow$	$computePath(u)$
$\mathbf{R_{EB}}(u)$	:	$st_u = C \wedge \neg P\_correction(u) \wedge$	$\rightarrow$	$st_u := EB$
		$(abRoot(u) \vee st_{par_u} = EB)$		
$\mathbf{R_{EF}}(u)$	:	$st_u = EB \wedge (\forall v \in children(u) \mid st_v = EF)$	$\rightarrow$	$st_u := EF$
$\mathbf{R_I}(u)$	:	$P\_reset(u) \wedge (\forall v \in \Gamma(u) \mid st_v \neq C)$	$\rightarrow$	$st_u := I$
$\mathbf{R_R}(u)$	:	$(P\_reset(u) \vee st_u = I) \wedge (\exists v \in \Gamma(u) \mid st_v = C)$	$\rightarrow$	$computePath(u)$

3.1 Variables

In RSP, each process u maintains three variables: st_u, par_u, and d_u. Those three variables are constant for the root process[(ii)], r: $st_r = C$, $par_r = \perp$[(iii)], and $d_r = 0$. For each non-root process u, we have:

- $st_u \in \{I, C, EB, EF\}$, this variable gives the *status* of the process. I, C, EB, and EF respectively stand for *Isolated*, *Correct*, *Error Broadcast*, and *Error Feedback*. The two first states, I and C, are involved in the normal behavior of the algorithm, while the two last ones, EB and EF, are used during the correction mechanism. Precisely, $st_u = C$ (resp. $st_u = I$) means that u believes it is in V_r (resp. not in V_r). The meaning of status EB and EF will be further detailed in Subsection 3.3.

- $par_u \in Lbl$, a *parent pointer*. If $u \in V_r$, par_u should designate a neighbor of u, referred to as its *parent*, and in a terminal configuration, the parent pointers exhibit a shortest path from u to r.

  Otherwise ($u \notin V_r$), the variable is meaningless.

- $d_u \in \mathbb{R}^+$, the *distance* value. If $u \in V_r$, then in a terminal configuration, d_u gives the weight of the shortest path from u to r.

  Otherwise ($u \notin V_r$), the variable is meaningless.

3.2 Normal Execution

Consider any configuration, where every process $u \neq r$ satisfies $st_u = I$, and refer to such a configuration as a *normal initial configuration*. Each configuration reachable from a *normal initial configuration* is called a *normal configuration*, otherwise it is an *abnormal configuration*. Recall that $st_r = C$ in all configurations. Then, starting from a normal initial configuration, all processes in a connected component different from V_r are disabled forever. Focus now on the connected component V_r. Each neighbor u of r is enabled to execute $\mathbf{R_R}(u)$. A process eventually chooses r as parent by executing this rule, which in particular sets its status to C. Then, executions of rule $\mathbf{R_R}$ are asynchronously propagated in V_r until all its processes have status C: when a process u with status I finds one of its neighbor with status C it executes $\mathbf{R_R}(u)$, *i.e.* u takes status C and chooses as parent its neighbor v with status C such that $d_v + \omega(u, v)$ is minimum, d_u being updated accordingly. In parallel, rules $\mathbf{R_C}$ are executed to reduce the weight of the tree rooted at r: when a process u with status C can reduce d_u by selecting another neighbor with status C as parent, it chooses the one allowing to minimize d_u by executing $\mathbf{R_C}(u)$. Hence, eventually, the system reaches a terminal configuration, where the tree rooted at r is a shortest-path tree spanning all processes of V_r.

3.3 Error Correction

Assume now that the system is in an abnormal configuration. Thanks to the predicate $abRoot$, some non-root processes locally detect that their state is inconsistent with that of their neighbors. We call *abnormal roots* such processes. Informally (see Algorithm RSP for the formal definition), a process $u \neq r$ is an *abnormal root* if u is not isolated (*i.e.*, $st_u \neq I$) and satisfies one of the following four conditions:

[(ii)]We should emphasize that the use of constants at the root is not a limitation, rather it allows to simplify the design and proof of the algorithm. Indeed, these constants can be removed by adding a rule to correct all root's variables, if necessary, within a single step.

[(iii)]$\perp$ is a particular value which is different from any value in Lbl.

1. its parent pointer does not designate a neighbor,

2. its parent has status I,

3. its distance value d_u is inconsistent with the distance value of its parent, or

4. its status is inconsistent with the status of its parent.

Every non-root process u that is not an abnormal root satisfies one of the two following cases. Either u is *isolated*, *i.e.*, $st_u = I$, 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 Algorithm RSP for the formal definition). Consider a path $\mathcal{P} = u_1, \ldots, u_k$ (with $k \geq 1$) such that u_1 is either r or an abnormal root, and $\forall i, 1 \leq i < k, u_{i+1} \in children(u_i)$. $\mathcal{P}$ is acyclic and called a *branch* rooted at u_1. Hence, we define the normal tree $T(r)$ (resp. an abnormal tree $T(v)$, for any abnormal root v) as the set of all processes that belong to a branch rooted at r (resp. v).

Then, the goal is to remove all abnormal trees so that the system recovers a normal configuration. For each abnormal tree T, we have two cases. In the former case, the abnormal root u of T can join another tree T' using rule $\mathbf{R_C}(u)$, making T a subtree of T'. In the latter case, 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 trees, each of those is rooted at a previous child of u; and later u joins one of those (created) trees. Hence, the idea is to freeze T, before removing it. By freezing we mean assigning each member of the tree to a particular state, here EF, so that (1) no member v of the tree is allowed to execute $\mathbf{R_C}(v)$, and (2) no process w can join the tree by executing $\mathbf{R_R}(w)$. Once frozen, the tree can be safely deleted from its root to its leaves.

The freezing mechanism (inspired from [BCV03]) is achieved using the status EB and EF, and the rules $\mathbf{R_{EB}}$ and $\mathbf{R_{EF}}$. If a process is not involved into any freezing operation, then its status is I or C. Otherwise, it has status EB or EF and no neighbor can select it as its parent. These two latter states are actually used to perform a "Propagation of Information with Feedback" [Cha82, Seg83] in the abnormal trees. This is why status EB means "Error Broadcast" and EF means "Error Feedback". From an abnormal root, the status EB is broadcast down in the tree using rule $\mathbf{R_{EB}}$. Then, once the EB wave reaches a leaf, the leaf initiates a convergecast EF-wave using rule $\mathbf{R_{EF}}$. Once the EF-wave reaches the abnormal root, the tree is said to be *dead*, meaning that all processes in the tree have status EF and, consequently, no other process can join it. So, the tree can be safely deleted from its abnormal root toward its leaves. There is two possibilities for the deletion. If the process u to be deleted has a neighbor with status C, then it executes rule $\mathbf{R_R}(u)$ to directly join another "alive" tree. Otherwise, u becomes isolated by executing rule $\mathbf{R_I}(u)$, and u may join another tree later.

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

Actually, the freezing mechanism ensures that if a process is the root of an abnormal alive tree, it is in that situation since the initial configuration (see Lemma 4, page 12). The polynomial step complexity mainly relies on this strong property.

3.4 Example

An example of synchronous execution of RSP is given in Figure 1. We consider the network topology given on the top left of the figure. The names $v_1, \ldots, v_{10}$ are only given to ease the explanation (recall that we consider semi-anonymous networks where only the root r is distinguished). The network contains eleven processes divided into two connected components. Let v_i be a process. In the synchronous execution described from configuration a) to configuration m), the color of v_i indicates its status st_{v_i}, according to the legend on the top of the figure. The number next to v_i gives its distance value, d_{v_i}. If there is an arrow outgoing from v_i, this arrow designates the neighbor u of v_i pointed as parent, *i.e.*, $par_{v_i} = u$. Otherwise, this means that $par_{v_i} \notin \Gamma(v_i)$.

In the initial configuration a), there are two abnormal roots: v_2 and v_9, indeed $par_{v_2} \notin \Gamma(v_2)$ and $par_{v_9} \notin \Gamma(v_9)$. The status of v_2 is already equal to EB and this value should be broadcast down in its subtree. In contrast, v_9 has status C and, consequently, should initiate the broadcast of EB. Note also that v_{10} can reduce its distance value by modifying its parent pointer. Hence, in the step a) $\mapsto$ b), v_9 takes status EB (rule $\mathbf{R_{EB}}(v_9)$), v_{10} selects v_9 as parent (rule $\mathbf{R_C}(v_{10})$), and finally v_3 the unique child of v_2 takes status EB (rule $\mathbf{R_{EB}}(v_3)$).

In the step b) $\mapsto$ c), EB is propagated down the two abnormal trees: v_5, v_7, v_8, and v_{10} execute $\mathbf{R_{EB}}$. In configuration c), the value EB has reached three leaves: v_5, v_7, and v_{10}. These processes are then enabled to initiate a convergecast EF-wave. Hence, in the step c) $\mapsto$ d), v_5, v_7, and v_{10} execute $\mathbf{R_{EF}}$, while the last leaf v_4 takes status EB ($\mathbf{R_{EB}}(v_4)$).

In configuration d), all children of v_9 have status EF, so v_9 is enabled to take status EF too ($\mathbf{R_{EF}}(v_9)$). In contrast, v_3 should wait until its child v_8 takes status EF. Hence, in the step d) $\mapsto$ e), v_9 takes status EF ($\mathbf{R_{EF}}(v_9)$), its abnormal tree becomes frozen, while the last leaf v_4 of the second abnormal tree initiates a convergecast EF-wave (rule $\mathbf{R_{EF}}(v_4)$).

In the step e) $\mapsto$ f), v_9 leaves its tree and becomes isolated by rule $\mathbf{R_I}(v_9)$, while v_8 takes status EF by $\mathbf{R_{EF}}(v_8)$. Since all its children have now status EF, v_3 can take status EF by $\mathbf{R_{EF}}(v_3)$ in step f) $\mapsto$ g), while $v5$ and v_{10} become isolated by rule $\mathbf{R_I}$ in the same step. Remark then that in g), all processes in the connected component $\{v_5, v_9, v_{10}\}$ are isolated and, since r is not part of this component, they are disabled forever. In the step g) $\mapsto$ h), the abnormal root v_2 of the remaining abnormal tree takes status EF ($\mathbf{R_{EF}}(v_2)$). So, the abnormal tree rooted at v_2 is frozen in configuration h). In the step h) $\mapsto$ i), v_2 leaves its tree and becomes isolated by rule $\mathbf{R_I}(v_2)$. Then, v_3 becomes isolated in step i) $\mapsto$ j) (rule $\mathbf{R_I}(v_3)$). In step j) $\mapsto$ k), v_8 becomes isolated (rule $\mathbf{R_I}(v_8)$), while v_7 joins the normal tree (the tree rooted at r) by rule $\mathbf{R_R}(v_7)$. In the last two steps, v_2, v_3, v_8, and then v_4 successively join the normal tree by rule $\mathbf{R_R}$, and configuration m) is terminal.

4   Correctness and Step Complexity of Algorithm RSP

4.1 Definitions

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

Definition 2 (Abnormal Root) *Every process $u \neq r$ that satisfies $abRoot(u)$ is said to be an* abnormal root.

Definition 3 (Alive Abnormal Root) *A process $u \neq r$ 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).*
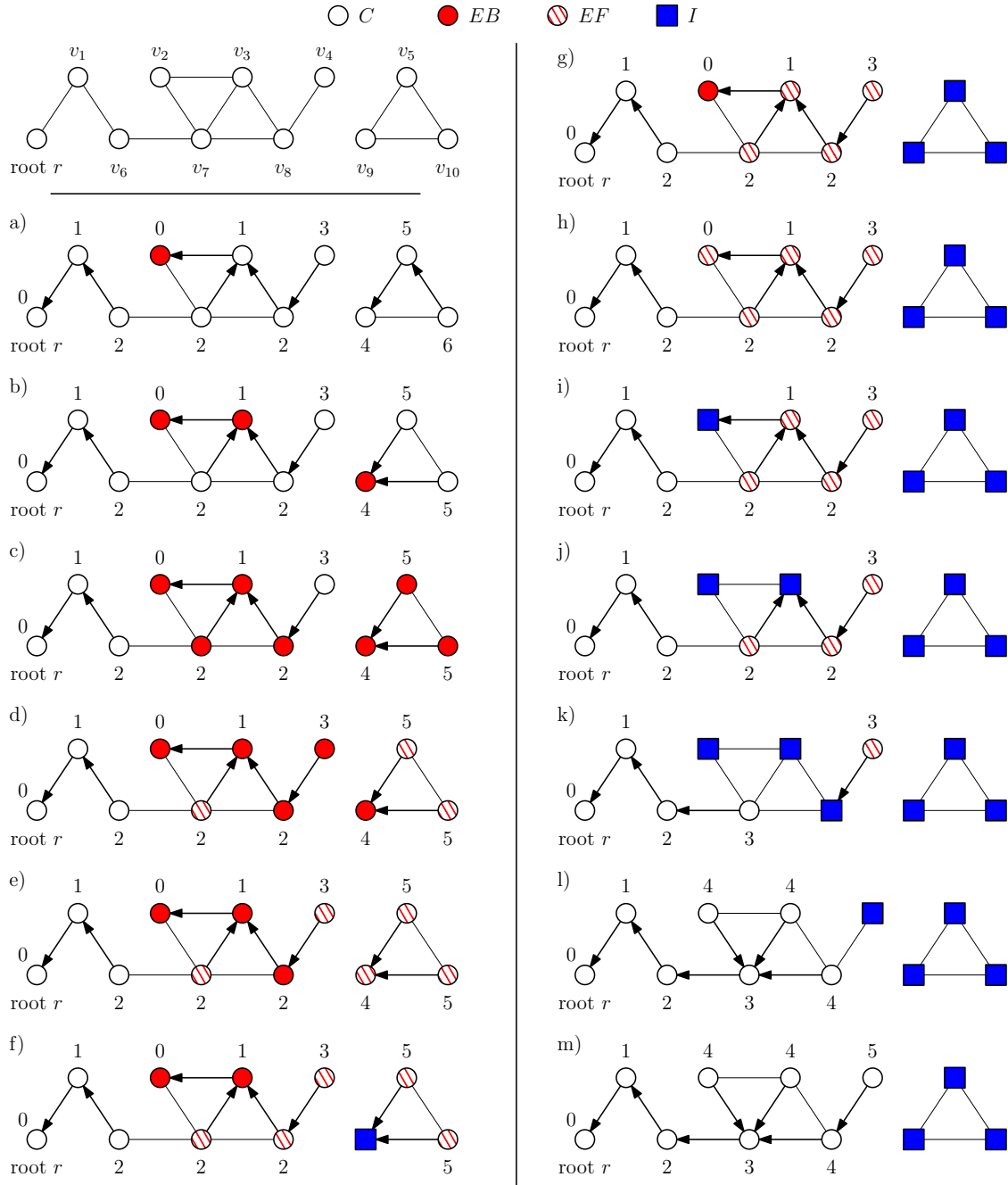
Fig. 1: A synchronous execution of RSP

Definition 4 (Branch) *A* branch *is a sequence of processes* $v_1, \cdots, v_k$ *for some integer* $k \geq 1$, *such that* v_1 *is* r *or an abnormal root and, for every* $1 \leq i < k$, *we have* $v_{i+1} \in children(v_i)$. *The process* v_i *is said to be at* depth i *and* $v_i, \cdots, v_k$ *is called a* sub-branch. *If* $v_1 \neq r$, *the branch is said to be* illegal, *otherwise, the branch is said to be* legal.

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

Definition 5 (Legitimate State) *A process* u *is said to be in a* legitimate state *if* u *satisfies one of the following three conditions:*

1. $u = r$,

2. $u \neq r$, $u \in V_r$, $st_u = C$, $d_u = d(u, r)$, *and* $d_u = d_{par_u} + \omega(u, par_u)$, *or*

3. $u \notin V_r$ *and* $st_u = I$.

Observation 2 *Every process* $u \neq r$ *such that* $st_u = C$ *and* $d_u \neq d_{par_u} + \omega(u, par_u)$ *is enabled.*

Definition 6 (Legitimate Configuration) *A* legitimate configuration *is any configuration where every process is in a legitimate state. We denote by* $\mathcal{LC}_{\mathsf{RSP}}$ *the set of all legitimate configurations of Algorithm* RSP.

Let γ be a configuration. Let $T_\gamma = (V_r, E_{T_\gamma})$ be the subgraph, where $E_{T_\gamma} = \{\{p, q\} \in E \mid p \in V_r \setminus \{r\} \wedge par_p = q\}$. By Definition 5 (point 2), we deduce the following observation.

Observation 3 *In every legitimate configuration* γ, T_γ *is a shortest-path tree spanning all processes of* V_r.

4.2 Partial Correctness

We now prove that the set of terminal configurations is exactly the set of legitimate configurations. We start by proving the following intermediate statement.

Lemma 1 *In any terminal configuration, every process has either status* I *or* C.

Proof: This is trivially true for the root process, r. Assume that there exists a non-root process with status EB in a terminal configuration γ. Consider the non-root process u with status EB having the largest distance value d_u in γ. In γ, no process v with status C can be a child of u, otherwise either $\mathrm{R_{EB}}$ or $\mathrm{R_C}$ is enabled at v in γ, a contradiction. Moreover, by maximality of d_u, u cannot have a child with status EB in γ. Therefore, in γ process u has no child or it has only children with status EF, and thus rule $\mathrm{R_{EF}}$ is enabled at u, a contradiction. Thus, every process has status C, I, or EF in γ.

Assume now that there exists a non-root process with status EF in a terminal configuration γ. Consider the process u with status EF having the smallest distance value d_u in γ. By construction, u is an abnormal root in γ. So, either $\mathrm{R_I}$ or $\mathrm{R_R}$ is enabled at u in γ, a contradiction. □

The next lemma, Lemma 2, deals with the connected components that do not contain r, if any. Then, Lemma 3 deals with the connected component V_r.

Lemma 2 *In any terminal configuration, every process that does not belong to* V_r *is in a legitimate state.*

Proof: Consider, by contradiction, that there exists a process u that belongs to the connected component CC other than V_r which is not in a legitimate state in some terminal configuration γ. By definition, u is not the root, moreover it has status C in γ, by Lemma 1. So, consider the process v of CC with status C having the smallest distance value d_v in γ. By construction, v is an abnormal root in γ. Thus, rule R_{EB} is enabled at v in γ, a contradiction. $\Box$

Lemma 3 *In any terminal configuration, every process of V_r is in a legitimate state.*

Proof: Assume, by contradiction, that there exists a terminal configuration γ where at least one process in the connected component V_r is not in a legitimate state.

Assume also that there exists some process of V_r that has status I in γ. Consider now a process u of V_r such that in γ, u has status I and at least one of its neighbors has status C. Such a process exists because no process has status EB or EF in γ (Lemma 1), but at least one process of V_r has status C, namely r. Then, R_R is enabled at u in γ, a contradiction. So, every process in V_r must have status C in γ. Moreover, for all processes in V_r, we have $d_u = d_{par_u} + \omega(par_u, u)$ in γ, otherwise R_C is enabled at some process of V_r in γ.

Assume now that there exists a process u such that $d_u < d(u, r)$ in γ. Consider a process u of V_r having the smallest distance value d_u among the processes in V_r such that $d_u < d(u, r)$ in γ. By definition, $u \neq r$ and we have $d_u > d_{par_u}$ in γ, so $d_{par_u} \geq d(par_u, r)$ in γ. Hence, we can conclude that $d_u \geq d(u, r)$ in γ, a contradiction. So, every process u in V_r satisfies $d_u \geq d(u, r)$ in γ.

Finally, assume that there exists a process u such that $d_u > d(u, r)$ in γ. Consider a process u in V_r having the smallest distance to r among the processes in V_r such that $d_u > d(u, r)$ in γ. By definition, $u \neq r$ and there exists some process v in $\Gamma(u)$ such that $d(u, r) = d(v, r) + \omega(u, v)$ in γ. Thus, we have $d_v = d(v, r)$ in γ. So, R_C is enabled at u in γ, a contradiction. $\Box$

After noticing that any legitimate configuration is a terminal one (by construction of the algorithm), we deduce the following corollary from the two previous lemmas.

Corollary 1 *For every configuration γ, γ is terminal if and only if γ is legitimate.*

4.3 Termination

In this section, we establish that every execution of Algorithm RSP under a distributed unfair daemon is finite. Furthermore, we compute the following bound on the number of steps of every execution: $[W_{\max} n_{\max CC}{}^3 + (3 - W_{\max}) n_{\max CC} + 3](n - 1)$, where n is the number of processes, $W_{\max}$ is the maximum weight of an edge, and $n_{\max CC}$ is the maximum number of non-root processes in a connected component, when all weights are strictly positive integers.

Lemma 4 *No alive abnormal root is created along any execution.*

Proof: Let $\gamma \mapsto \gamma'$ be a step. Let u be a non-root process that is not an *alive abnormal root* in γ, and let v be the process such that $par_u = v$ in γ'. If the status of u is EF or I in γ', then u is not an alive abnormal root in γ'. So, let us assume now that the status of u is either EB or C in γ'.

Consider then the case where u has status EB in γ'. The only rule u can execute in $\gamma \mapsto \gamma'$ is R_{EB}. So, $st_u \in \{C, EB\}$ in γ. Moreover, whether u executes R_{EB} or not, $par_u = v$ in γ. Since $st_u \in \{C, EB\}$ and u is not an alive abnormal root in γ, we can deduce that u is not an abnormal root in γ (whether dead or alive). So, if $st_u = EB$ in γ, then $st_v = EB$ in γ too. Otherwise, u has status C in γ while not being

an abnormal root in γ: it executes $R_{EB}(u)$ in $\gamma \mapsto \gamma'$ because $st_v = EB$ in γ. Hence, in either case v has status EB in γ, and this in particular means that $v \neq r$ (this status does not exist for r). Moreover, u belongs to $children(v)$ in γ (again because $par_u = v$ and u is not an abnormal root in γ). So, v is not enabled in γ and $u \in children(v)$ remains true in γ'. Hence, we can conclude that u is still not an alive abnormal root in γ'.

Consider now the other case, *i.e.*, u has status C in γ'. During $\gamma \mapsto \gamma'$, the only rules that u may execute are R_R or R_C. If u executes R_R or R_C, we have $st_v = C$ in γ (because it is a requirement to execute any of these rules) and consequently, the only rules that v may execute in $\gamma \mapsto \gamma'$ are R_C or R_{EB}. Otherwise (*i.e.*, u does not execute any rule in $\gamma \mapsto \gamma'$), $par_u = v$ and $st_u = C$ already hold in γ. In this case, u being not an alive abnormal root and $st_u = C$ in γ implies that $u \in children(v)$ and thus $st_v \in \{C, EB\}$ in γ, which further implies that the only rules that v may execute in $\gamma \mapsto \gamma'$ in this case are R_C or R_{EB}. Thus, in either case, during $\gamma \mapsto \gamma'$, v either takes the status EB, decreases its distance value, or does not change the value of its variables. Consequently, u belongs to $children(v)$ in γ', which prevents u from being an alive abnormal root in γ'. □

Let $AAR(\gamma)$ be the set of alive abnormal roots in any configuration γ. From the previous lemma, we know that, for every step $\gamma \mapsto \gamma'$, we have $AAR(\gamma') \subseteq AAR(\gamma)$ (precisely, for every process u and every step $\gamma \mapsto \gamma'$, $u \notin AAR(\gamma) \Rightarrow u \notin AAR(\gamma')$). So, we can use the notion of *u-segment* (inspired from [ACD$^+$17]) to bound the total number of steps in an execution.

Definition 7 (u-Segment) *Let u be any non-root process. Let $e = \gamma_0, \gamma_1, \cdots$ be an execution.*

If there is no step $\gamma_i \mapsto \gamma_{i+1}$ in e, where there is a non-root process in V_u which is an alive abnormal root in γ_i, but not in γ_{i+1}, then the first *u-segment of e is e itself and there is no other u-segment.*

Otherwise, let $\gamma_i \mapsto \gamma_{i+1}$ be the first step of e, where there is a non-root process in V_u which is an alive abnormal root in γ_i, but not in γ_{i+1}. The first *u-segment of e is the prefix $\gamma_0, \cdots, \gamma_{i+1}$. The* second *u-segment of e is the first u-segment of the suffix $\gamma_{i+1}, \gamma_{i+2}, \cdots$, and so forth.*

By Lemma 4, we have

Observation 4 *For every non-root process u, for every execution e, e contains at most $n_{maxCC} + 1$ u-segments, because there are initially at most n_{maxCC} alive abnormal roots in V_u.*

Lemma 5 *Let u be any non-root process. During a u-segment, if u executes the rule R_{EF}, then u does not execute any other rule in the remaining of the u-segment.*

Proof: Let seg_u be a u-segment. Let s_1 be a step of seg_u in which u executes R_{EF}. Let s_2 be the next step in which u executes its next rule. (If s_1 or s_2 do not exist, then the lemma trivially holds for seg_u.) Just before s_1, all branches containing u have an alive abnormal root, namely the non-root process v at depth 1 in any of these branches. (Note that we may have $v = u$.) On the other hand, just before s_2, u is the dead abnormal root of all branches it belongs to. This implies that v must have executed the rule R_{EF} in the meantime and thus is not an alive abnormal root anymore when the step s_2 is executed. Therefore, s_1 and s_2 belong to two distinct u-segments of the execution. □

Corollary 2 *Let u be a non-root process. The sequence of rules executed by u during a u-segment belongs to the following language:* $(R_I + \varepsilon)(R_R + \varepsilon)R_C{}^*(R_{EB} + \varepsilon)(R_{EF} + \varepsilon)$.

We use the notion of *maximal causal chain* to further analyze the number of steps in a u-segment.

Definition 8 (Maximal Causal Chain) *Let u be a non-root process and seg_u be any u-segment. A maximal causal chain of seg_u rooted at $u_0 \in V_u$ is a maximal sequence of actions $a_1, a_2, \cdots, a_k$ executed in seg_u such that the action a_1 sets par_{u_1} to $u_0 \in V_u$ not later than any other action by u_0 in seg_u, and for all $2 \leq i \leq k$, the action a_i sets par_{u_i} to u_{i-1} after the action a_{i-1} but not later than u_{i-1}'s next action.*

Observation 5

- *An action a_i belongs to a maximal causal chain if and only if a_i consists in a call to the macro $computePath$ by a non-root process.*

- *Only actions of Rules $\mathrm{R_R}$ and $\mathrm{R_C}$ contain the execution of $computePath$.*

Let u be a non-root process and seg_u be any u-segment. Let $a_1, a_2, \cdots, a_k$ be a maximal causal chain of seg_u rooted at u_0.

- *For all $1 \leq i \leq k$, a_i consists in the execution of $computePath$ by u_i (i.e., u_i executes the rule $\mathrm{R_R}$ or $\mathrm{R_C}$) where $u_i \in V_u$.*

- *Denote by $ds_{\mathsf{seg}_u,v}$ the distance value of process v at the beginning of seg_u. For all $1 \leq i \leq k$, a_i sets d_{u_i} to $ds_{\mathsf{seg}_u,u_0} + \sum_{j=1}^{j=i} w(u_j, u_{j-1})$, where u_i is the process that executes a_i.*

For the next lemmas and theorems, we recall that $n_{\mathrm{maxCC}} \leq n - 1$ is the maximum number of non-root processes in a connected component of G.

Lemma 6 *Let u be a non-root process. All actions in a maximal causal chain of a u-segment are caused by different non-root processes of V_u. Moreover, an execution of $computePath$ by some non-root process v never belongs to any maximal causal chain rooted at v.*

Proof: First note that any rule $\mathrm{R_C}$ executed by a process v makes the value of d_v decrease.

Assume now, by the contradiction, that there exists a process v such that, in some maximal causal chain $a_1, a_2, \cdots, a_k$ of a u-segment, v is used as parent in some action a_i and executes the action a_j, with $j > i$. The value of d_v is strictly larger just after the action a_j than just before the action a_i. This implies that process v must have executed the rule $\mathrm{R_R}$ in the meantime. So, a_i and a_j are executed in two different u-segments by Corollary 2 and the fact that v has status C just before the action a_i. Consequently, they do not belong to the same maximal causal chain, a contradiction.

Therefore, all actions in a maximal causal chain are caused by different processes, and a process never executes an action in a maximal causal chain it is the root of. As all actions in a maximal causal chain are executed by processes in the same connected component, we are done.   □

Definition 9 ($S_{\mathsf{seg}_u,v}$) *Given a non-root process u and a u-segment seg_u, we define $S_{\mathsf{seg}_u,v}$ as the set of all the distance values obtained after executing an action belonging to any maximal causal chain of seg_u rooted at process v ($v \in V_u$).*

Note that, from Observation 5 and Lemma 6, we have the following observation:

Observation 6 *The size of the set $S_{\mathsf{seg}_u,v}$ is bounded by a function of the number of processes in V_u.*

Lemma 7 *Given a non-root process u and a u-segment seg_u, if the size of $S_{\mathsf{seg}_u,v}$ is bounded by X for all process $v \in V_u$, then the number of computePath executions done by u in seg_u is bounded by $X(n_{maxCC} - 1)$.*

Proof: Except possibly the first, all *computePath* executions done by a u in a u-segment seg_u are done through the rule $\mathrm{R_C}$. For all these, the variable d_u is always decreasing. Therefore, all the values of d_u obtained by the *computePath* executions done by u are different. By definition of $S_{\mathsf{seg}_u,v}$ and by Lemma 6, all these values belong to the set $\bigcup_{v \in V_u \setminus \{u\}} S_{\mathsf{seg}_u,v}$, which has size at most $X(n_{\max CC} - 1)$. □

By definition, each step contains at least one action, made by a non-root process. Let u be any non-root process. Assume that, in any u-segment seg_u, the size of $S_{\mathsf{seg}_u,v}$ is bounded by X for all process $v \in V_u$. So, the number of step of u in seg_u is bounded by $X(n_{\max CC} - 1) + 3$, by Lemma 7 and Corollary 2. Moreover, recall that each execution contains at most $n_{\max CC} + 1$ u-segments (Observation 4). So, u executes in at most $X n_{\max CC}{}^2 + 3 n_{\max CC} - X + 3$ steps. Finally, as u is an arbitrary non-root process and there are $n - 1$ non-root processes, follows.

Theorem 1 *If the size of $S_{\mathsf{seg}_u,v}$ is bounded by X for all non-root process u, for all u-segment seg_u, and for all process v in V_u, then the total number of steps during any execution, is bounded by $(X n_{maxCC}{}^2 + 3 n_{maxCC} - X + 3)(n - 1)$.*

Let $\mathtt{W}_{\max} = \max_{\{u,v\} \in E} \omega(u, v)$. If all weights are strictly positive integers, then the size of any $S_{\mathsf{seg}_u,v}$, where u is a non-root process and $v \in V_u$, is bounded by $\mathtt{W}_{\max} n_{\max CC}$, because $S_{\mathsf{seg}_u,v} \subseteq [ds_{\mathsf{seg}_u,v} + 1, ds_{\mathsf{seg}_u,v} + \mathtt{W}_{\max}(n_{cc} - 1)]$, where $n_{cc} \leq n_{\max CC} + 1$ is the number of processes in V_u. Hence, we deduce the following theorem from Theorem 1, Observation 6, and Corollary 1.

Theorem 2 *Algorithm* RSP *is silent self-stabilizing under the distributed unfair daemon for the set $\mathcal{LC}_{\mathsf{RSP}}$ and, when all weights are strictly positive integers, its stabilization time in steps is at most $[\mathtt{W}_{\max} n_{maxCC}{}^3 + (3 - \mathtt{W}_{\max}) n_{maxCC} + 3](n - 1)$, i.e., $O(\mathtt{W}_{\max} n_{maxCC}{}^3 n)$.*

If all edges in G have the same weight w, then the size of $S_{\mathsf{seg}_u,v}$, where u is a non-root process and $v \in V_u$, is bounded by $n_{\max CC}$. Indeed, in such a case, we have $S_{\mathsf{seg}_u,v} \subset \{ds_{\mathsf{seg}_u,v} + i.w \mid 1 \leq i \leq n_{cc} - 1\}$, where $n_{cc} \leq n_{\max CC} + 1$ is the number of processes in V_u. Hence, we obtain the following corollary.

Corollary 3 *If all edges have the same weight, then the stabilization time in steps of Algorithm* RSP *is at most $(n_{maxCC}{}^3 + 2 n_{maxCC} + 3)(n - 1)$, which is less than or equal to n^4 for all $n \geq 2$.*

5   Round Complexity of Algorithm RSP

We now prove that every execution of Algorithm RSP lasts at most $3 n_{\max CC} + D$ rounds, where $n_{\max CC}$ is the maximum number of non-root processes in a connected component and D is the hop-diameter of the connected component containing r, V_r.

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

Lemma 8 *Let $i \in \mathbb{N}^*$. Starting from the beginning of round i, there does not exist any process both in state C and at depth less than i in an illegal branch.*

Proof: We prove this lemma by induction on i. The base case ($i = 1$) is vacuum, so we assume that the lemma holds for some integer $i \geq 1$. From the beginning of round i, no process can ever choose a parent

which is at depth smaller than i in an illegal branch because those processes will never have status C, by induction hypothesis. Moreover, no process with status C can have its depth decreasing to i or smaller by an action of one of its ancestors at depth smaller than i, because these processes have status EB and have at least one child not having status EF. Thus, they cannot execute any rule. Therefore, no process can take state C at depth smaller or equal to i in an illegal branch.

Consider any process u with status C at depth i in an illegal branch at the beginning of the round i. $u \neq r$. Moreover, by induction hypothesis, u is an abnormal root, or the parent of u is not in state C (*i.e.*, it is in the state EB). During round i, u will execute rule R_{EB} or R_C and thus either switch to state EB or join another branch at a depth greater than i. This concludes the proof of the lemma. $\square$

Corollary 4 *After at most n_{maxCC} rounds, the system is in a configuration from which no process in any illegal branch has status C forever.*

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

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

Lemma 9 *Let $i \in \mathbb{N}^*$. Starting from the beginning of round $n_{maxCC} + i$, there does not exist any process at depth larger than $n_{maxCC} - i + 1$ in an illegal branch having the status EB.*

Proof: We prove this lemma by induction on i. The base case ($i = 1$) is vacuum (by Observation 1), so we assume that the lemma holds for some integer $i \geq 1$. At the beginning of round $n_{maxCC} + i$, no process at depth larger than $n_{maxCC} - i + 1$ has the status EB (by induction hypothesis) or status C (by Corollary 4). Therefore, processes with status EB at depth $n_{maxCC} - i + 1$ in an illegal branch can execute the rule R_{EF} at the beginning of round $n_{maxCC} + i$. These processes will thus all execute within round $n_{maxCC} + i$ (they cannot be neutralized as no children can connect to them). We conclude the proof by noticing that, from Corollary 4, once round n_{maxCC} has terminated, any process in an illegal branch that executes either gets status EF, or will be outside any illegal branch forever. $\square$

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

Lemma 10 *Let $i \in \mathbb{N}^*$. Starting from the beginning of round $2n_{maxCC} + i$, there does not exist any process at depth larger than $n_{maxCC} - i + 1$ in an illegal branch.*

Proof: We prove this lemma by induction on i. The base case ($i = 1$) is vacuum (by Observation 1), so we assume that the lemma holds for some integer $i \geq 1$. By induction hypothesis, at the beginning of round $2n_{maxCC} + i$, no process is at depth larger than or equal to $n_{maxCC} - i + 1$ in an illegal branch. All processes in an illegal branch have the status EF. So, at the beginning of round $2n_{maxCC} + i$, any abnormal root satisfies the predicate P_reset, they are enabled to execute either R_I, or R_R. So, all abnormal roots at the beginning of the round $2n_{maxCC} + i$ 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 4, no process can join an illegal tree during the round $2n_{maxCC} + i$. $\square$

Corollary 5 *After at most round $3n_{maxCC}$, there are no illegal branches forever.*

Note that in any connected component that does not contain the root r, there is no legal branch. Then, since the only way for a process to be in no branch is to have status I, we obtain the following corollary.

Corollary 6 *For any connected component H other than V_r, after at most $3n_{maxCC}$ rounds, every process of H is in a legitimate state forever.*

In the connected component V_r, Algorithm RSP may need additional rounds to propagate the correct distances to r. In the next lemma, we use the notion of hop-distance to r defined below.

Definition 10 (Hop-Distance and Hop-Diameter) *A process u is said to be at* hop-distance k from v if *the minimum number of edges in a shortest path from u to v is k.*

The hop-diameter *of a graph G (resp. of a connected component H of the graph G) is the maximum hop-distance between any two nodes of G (resp. of H).*

Lemma 11 *Let $i \in \mathbb{N}$. In every execution of Algorithm* RSP*, starting from the beginning of round $3n_{maxCC} + i$, every process at hop-distance at most i from r is in a legitimate state.*

Proof: We prove this lemma by induction on i. First, by definition, the root r is always in a legitimate state, so the base case ($i = 0$) trivially holds. Then, after at most $3n_{maxCC}$ rounds, every process either belongs to a legal branch or has status I (by Corollary 5), thus any non-isolated process $v \in V_r$ always stores a distance d such that $d \geq d(v, r)$, its actual weighted distance to r. By induction hypothesis, every process at hop-distance at most i from r has converged to a legitimate state within at most $3n_{maxCC} + i$ rounds. Therefore, at the beginning of round $3n_{maxCC} + i + 1$, every process v at hop-distance $i + 1$ from r which is not in a legitimate state is enabled for executing rule R_C. Thus, at the end of round $3n_{maxCC} + i + 1$, every process at hop-distance at most $i + 1$ from r is in a legitimate state (such processes cannot be neutralized during this round). Also, these processes will never change their state since there are no processes that can make them closer to r. □

Summarizing all the results of this section, we obtain the following theorem.

Theorem 3 *Every execution of Algorithm* RSP *lasts at most $3n_{maxCC} + D$ rounds, where n_{maxCC} is the maximum number of non-root processes in a connected component and D is the hop-diameter of the connected component containing r.*

6 Conclusion

In this paper, we have proposed a silent self-stabilizing algorithm for the DCDSPM problem. This algorithm is written in the composite atomicity model, assuming a distributed unfair daemon (the weakest scheduling assumption of the model). Its stabilization time in rounds is at most $3n_{maxCC} + D$, where n_{maxCC} is the maximum number of non-root processes in a connected component and D is the hop-diameter of V_r. Furthermore, if we additionally assume that edge weights are positive integers, then it stabilizes in a polynomial number of steps: namely, we exhibit a bound in $O(\mathbb{W}_{max} n_{maxCC}{}^3 n)$, where $\mathbb{W}_{max}$ is the maximum weight of an edge and n is the number of processes. To obtain this stabilization time polynomial in steps, the key idea was to freeze the growth of abnormal trees before removing them in a top-down manner. This freezing mechanism is implemented as a propagation of information with feedback in the tree. This technique is general. In particular, it can be used in other spanning tree or forest constructions.

The stabilization time is, by definition, evaluated from an arbitrary initial configuration, and so is drastically impacted by worst case scenarios. Now, in many cases, transient faults are sparse and their effect

may be superficial. For example, a topological change in a network commonly consists of a single link failure. Some specializations of self-stabilization, such as superstabilization [DH97], self-stabilization with service guarantee [JM14], or gradual stabilization [ADDP16] have been proposed to target recovery from such favorable cases as a performance issue. Proposing silent algorithms for the DCDSPM problem implementing one of these aforementioned stronger properties, while achieving polynomial step complexity, is an interesting perspective of our work.

References

[ACD+17]   Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing leader election in polynomial steps. *Inf. Comput.*, 254:330–366, 2017.

[ADDP16]   Karine Altisen, Stéphane Devismes, Anaïs Durand, and Franck Petit. Gradual stabilization under τ-dynamics. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, volume 9833 of *Lecture Notes in Computer Science*, pages 588–602. Springer, 2016.

[Afe13]   Yehuda Afek, editor. *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*. Springer, 2013.

[AGH90]   A Arora, MG Gouda, and T Herman. Composite routing protocols. In *the 2nd IEEE Symposium on Parallel and Distributed Processing (SPDP'90)*, pages 70–78, 1990.

[AGM+08]   I. Abraham, C. Gavoille, D. Malkhi, N. Nisan, and M. Thorup. Compact name-independent routing with minimum stretch. *ACM Transactions on Algorithms*, 4(3):37, 2008.

[BCV03]   Lélia Blin, Alain Cournier, and Vincent Villain. An improved snap-stabilizing PIF algorithm. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003*, volume 2704 of *Lecture Notes in Computer Science*, pages 199–214, San Francisco, CA, USA, June 24-25 2003. Springer.

[BDV07]   Doina Bein, Ajoy Kumar Datta, and Vincent Villain. Self-stabilizing local routing in ad hoc networks. *The Computer Journal*, 50(2):197–203, 2007.

[Bel58]   Richard Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[BPBRT10]   L. Blin, M. Potop-Butucaru, S. Rovedakis, and S. Tixeuil. Loop-free super-stabilizing spanning tree construction. In *the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10), Springer LNCS 6366*, pages 50–64, 2010.

[CDD+15]   Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. *J. Parallel Distrib. Comput.*, 81-82:11–23, 2015.

[CDPV06]   Alain Cournier, Stéphane Devismes, Franck Petit, and Vincent Villain. Snap-stabilizing depth-first search on arbitrary networks. *The Computer Journal*, 49(3):268–280, 2006.

[CDV05]   Alain Cournier, Stéphane Devismes, and Vincent Villain. A snap-stabilizing dfs with a lower space requirement. In *Symposium on Self-Stabilizing Systems*, pages 33–47. Springer, 2005.

[CDV09]   Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), 2009.

[CG02]   J. A. Cobb and M. G. Gouda. Stabilization of general loop-free routing. *Journal of Parallel and Distributed Computing*, 62(5):922–944, 2002.

[CH09]   J. A. Cobb and C.-T. Huang. Stabilization of maximal-metric routing without knowledge of network size. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 306–311. IEEE, 2009.

[Cha82]   Ernest J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.

[Cou09]   Alain Cournier. A new polynomial silent stabilizing spanning-tree construction algorithm. In *International Colloquium on Structural Information and Communication Complexity*, pages 141–153. Springer, 2009.

[CRV11]   Alain Cournier, Stephane Rovedakis, and Vincent Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. In *the 15th International Conference on Principles of Distributed Systems (OPODIS'11), Springer LNCS 7109*, pages 159–174, 2011.

[CS94]   Srinivasan Chandrasekar and Pradip K Srimani. A self-stabilizing distributed algorithm for all-pairs shortest path problem. *Parallel Algorithms and Applications*, 4(1-2):125–137, 1994.

[CYH91]   NS Chen, HP Yu, and ST Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[DDL12]   Ajoy Kumar Datta, Stéphane Devismes, and Lawrence L. Larmore. Brief announcement: Self-stabilizing silent disjunction in an anonymous network. In *the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'12), Springer LNCS 7596*, pages 46–48, 2012.

[DDL13]   Ajoy Kumar Datta, Stéphane Devismes, and Lawrence L. Larmore. Self-stabilizing silent disjunction in an anonymous network. In *14th International Conference on Distributed Computing and Networking (ICDCN 2013), Springer LNCS 7730*, pages 148–160, 2013.

[DGS99]   Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.

[DH97]   Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[Dij74]   Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[DIJ16]   Stéphane Devismes, David Ilcinkas, and Colette Johnen. Self-stabilizing disconnected components detection and rooted shortest-path tree maintenance in polynomial steps. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016*, volume 70 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[DIM93]   S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only Read/Write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[DJ16]   Stéphane Devismes and Colette Johnen. Silent self-stabilizing {BFS} tree algorithms revisited. *Journal of Parallel and Distributed Computing*, 97:11 – 23, 2016.

[DLP10]   Ajoy Kumar Datta, Lawrence L. Larmore, and Hema Piniganti. Self-stabilizing leader election in dynamic networks. In *the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10), Springer LNCS 6366*, pages 35–49, 2010.

[DLV11a]   Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An o(n)-time self-stabilizing leader election algorithm. *jpdc*, 71(11):1532–1544, 2011.

[DLV11b]   Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011.

[Dol00]   Shlomi Dolev. *Self-stabilization*. MIT Press, March 2000.

[FJ56]   Lester R. Ford Jr. Network flow theory. *RAND Corporation*, (Paper P-923), August 14 1956.

[Ga16]   Christian Glacet and Nicolas Hanusse and. Disconnected components detection and rooted shortest-path tree maintenance in networks - extended version. Technical report, LaBRI, CNRS UMR 5800, 2016.

[GGHI13]   Cyril Gavoille, Christian Glacet, Nicolas Hanusse, and David Ilcinkas. On the communication complexity of distributed name-independent routing schemes. In *the 27th International Symposium on Distributed Computing (DISC'13), Springer LNCS 8205*, pages 418–432, 2013.

[GHIJ14]   Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks. In *the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'14), Springer LNCS 8736*, pages 120–134, 2014.

[Gä03]   Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, Swiss Federal Institute of Technolog (EPFL), 2003.

[HC92]   Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.

[Hed88]   Charles L Hedrick. Routing information protocol, 1988.

[HL02]   Tetz C Huang and Ji-Cherng Lin. A self-stabilizing algorithm for the shortest path problem in a distributed system. *Computers & Mathematics with Applications*, 43(1):103–109, 2002.

[Hua05a]   Tetz C. Huang. A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity. *Journal of Computer System Sciences*, 71(1):70–85, 2005.

[Hua05b]   Tetz C Huang. A self-stabilizing algorithm for the shortest path problem assuming the distributed demon. *Computers & Mathematics with Applications*, 50(5–6):671 – 681, 2005.

[JM14]   Colette Johnen and Fouzi Mekhaldi. Self-stabilizing with service guarantee construction of 1-hop weight-based bounded size clusters. *Journal of Parallel and Distributed Computing*, 74(1):1900–1913, 2014.

[JT03]   C. Johnen and S. Tixeuil. Route preserving stabilization. In *the 6th International Symposium on Self-stabilizing System (SSS'03), Springer LNCS 2704*, pages 184–198, 2003.

[KK05]   Adrian Kosowski and Lukasz Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In *6th International Conference Parallel Processing and Applied Mathematics, (PPAM'05), Springer LNCS 3911*, pages 75–82, 2005.

[LGW04]   Alberto Leon-Garcia and Indra Widjaja. *Communication Networks*. McGraw-Hill, Inc., New York, NY, USA, 2 edition, 2004.

[RLH06]   Y Rekhter, T Li, and S Hares. Rfc 4271: Border gateway protocol 4, 2006.

[Seg83]   Adrian Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.

[SK87]   M Sloman and J Kramer. *Distributed systems and computer networks.* Prentice Hall, 1987.

[Tel01]   G Tel. *Introduction to distributed algorithms*. Cambridge University Press, Cambridge, UK, Second edition 2001.