Disconnected components detection and rooted shortest-path tree maintenance in networks[☆]

Christian Glacet[a], Nicolas Hanusse[b], David Ilcinkas[b], Colette Johnen[a]

[a] *Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France*
[b] *CNRS, LaBRI, UMR 5800, F-33400 Talence, France*

Abstract

Many articles deal with the problem of maintaining a rooted shortest-path tree. However, after some edge deletions, some nodes can be disconnected from the connected component V_r of some distinguished node r. In this case, an additional objective is to ensure the detection of the disconnection by the nodes that no longer belong to V_r. We present a detailed analysis of a silent self-stabilizing algorithm. We prove that it solves this more demanding task in anonymous weighted networks with the following additional strong properties: it runs without any knowledge on the network and under the *unfair* daemon, that is without any assumption on the asynchronous model. Moreover, it terminates in less than $2n + D$ rounds for a network of n nodes and hop-diameter D.

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

1. Introduction

Routing algorithms using the computation of distance/path vectors, like RIP (*Routing information protocol*) or BGP (*Border Gateway Protocol*), are based on the construction of shortest-path trees. For any destination r, a shortest-path tree rooted at r is implicitly built by the routing scheme. Because of the dynamics of the network, it may happen that the network is disconnected. Routing to node r is only guaranteed from the nodes that belong to the same component as r, namely V_r. For the other nodes, one should remove, in the routing tables, information to reach r in order to prevent routing messages that will anyway never reach r, and thus to save some bandwidth. A legitimate configuration is characterized by the fact that every node that belongs to V_r knows a route to r and every node not in V_r detects that r is not in its own component. The difficulty of converging toward a legitimate configuration is called, in this context, the count-to-infinity problem [2]: for nodes that do not belong to V_r, some control messages keep on being exchanged infinitely in order to find a path to r. At the same time, the updates of routing tables for nodes belonging to V_r should be done as quickly as possible.

[☆] A preliminary version of this work was presented to SSS'14 [1].

Email addresses: christian.glacet@gmail.com (Christian Glacet), nicolas.hanusse@labri.fr (Nicolas Hanusse), david.ilcinkas@labri.fr (David Ilcinkas), johnen@labri.fr (Colette Johnen)

In practice, the most standard techniques consist in exchanging distance/path vectors periodically and in using some timers in order to guess if a node is still within V_r. However, the convergence is usually only guaranteed under some assumptions (i) on the asynchrony of the network and/or (ii) on some known upper bound on the diameter or the size of the network. The convergence towards a legitimate configuration can be often provided by self-stabilizing algorithms, for which the correctness is guaranteed for any initial configuration.

However, solutions of the literature dedicated to the maintenance of a BFS tree or shortest paths are mainly for connected networks. Using them, we still face the count-to-infinity problem in the disconnected components.

In the routing context, it is not always required to store information for every node. In compact routing schemes [3, 4], only some shortest-path trees completely spanning the connected components are built and need to be maintained. Given a set of roots $r_1, r_2, \ldots, r_k$, we aim at providing silent[1] self-stabilizing algorithms that both maintain a shortest-path tree toward each r_i, for nodes of V_{r_i}, and detect the nodes that no longer belong to V_{r_i}. In the following, we present an algorithm for a single root but our solution holds for any k. The identifiers of nodes do not need to be unique. Only r_i's identifiers should be different in order to distinguish the different roots. Thus, for $k = 1$, our self-stabilizing algorithm works in anonymous networks in the semi-uniform model: all nodes except one, the root, perform the same distributed algorithm.

In the asynchronous setting, the expressiveness of distributed algorithms is given by the scheduling of distributed processes. A given hypothesis on the scheduling is called a *daemon*. The highest level of asynchrony is called the *unfair daemon* since there is no restriction at all on the sequence of actions. In distributed computing, many algorithms works only for some assumption or restriction of daemons and it is always challenging to propose a solution for the unfair daemon. Having a protocol that works without any assumption on the scheduling can be very crucial when dealing with: (i) systems with very high inherent asynchrony (process speeds are not constant and/or processes have very heterogeneous speeds), e.g., ad-hoc networks; (ii) safety-critical systems, i.e., systems with low to no tolerance of (software) failures. In this latter case it is not acceptable to use a software that may crash, even if these crashes would happen under some unlikely scenarios. We will focus on unfair daemon for our study.

The performance of an asynchronous distributed algorithm is often measured by the time complexity expressed as a number of rounds. Informally, a *round* is the smallest fragment of time for which every distributed process can compute at least one execution step. Thus, rounds intuitively count the number of execution steps of the slowest process. Note that the number of steps is rarely analyzed in the literature. The first paper containing a self-stabilizing BFS construction with a bound on the number of steps was only published in 2009 [5] (17 years after the first paper presenting a self-stabilizing BFS construction: the Huang and Chen algorithm [6] in 1992). Moreover, in this 2009 paper, the algorithm is not silent, meaning that some useless actions can take place forever.

With an extra cost in terms of memory and a convergence time of $O(n)$ rounds for an n-node network, the asynchronous unison allows to transform any self-stabilizing algorithm designed under a weak daemon, the distributed fair daemon, into one dealing

[1] Eventually, the states of all nodes remain unchanged. This notion is also referred as "quiescent" in some contexts.

with the unfair daemon [7]. However, there are no guaranteed bounds on the number of steps. Thus, it is always difficult to provide a self-stabilizing algorithm working under the unfair daemon and providing theoretical bounds simultaneously on the number of rounds and the number of steps.

1.1. Related works

Self-stabilizing single-destination shortest-path constructions. The single-destination shortest-path problem is to find shortest paths from all vertices in the graph to a single destination vertex r. Edges can have weights and the length of a path corresponds to its sum of weights. The oldest distributed algorithms are inspired by the Bellman-Ford algorithm. In the articles dedicated to self-stabilizing algorithms in asynchronous networks, the difficulty is to find an algorithm that runs under the worst possible scenario. A scenario is a sequence of computational steps which is controlled by an adversary called *daemon*. In [8, 9], self-stabilizing algorithms for the single-destination shortest-path problem are presented; both protocols require a central daemon, that is only one process can be executed at each instant. In [10], Huang proves that the algorithms in [8, 9] also work under the unfair daemon, which is the most general daemon. However, no upper bounds on the time (rounds or number of execution steps) are given. The same author presents an algorithm under the read/write separate atomicity model (Dolev Model) in [11].

In [12, 13, 14], self-stabilizing algorithms for the single-destination shortest-path problem are presented; these algorithms ensure the loop-free property: after any edge weight changes, even during the rebuilding phase, there is always a path from any node to the destination. More generally, none of these articles provide tight bounds on the complexity of the convergence time in the most general asynchronous model, the unfair daemon, and the presented algorithms are not silent in the disconnected components.

Self-stabilizing breadth-first tree constructions. Whenever edges do not have any weight, shortest-path trees correspond to breadth-first trees. To our knowledge, this restriction does not help to get all the desirable guarantees. Chen et al. present the first self-stabilizing BFS tree construction in [15] under the central daemon. Huang et al. present the first self-stabilizing BFS tree construction in [6] under the unfair distributed daemon. In [15, 6], the exact network size has to be known by all nodes. Dolev, Israeli and Moran in [16] present the first self-stabilizing BFS spanning-tree construction algorithm under read/write atomicity.

Blin et al. in [17] present a universal transformer of self-stabilizing tree construction with any metric on semi-uniform networks to a loop-free super-stabilizing algorithm under the fair daemon. All these cited works assume that the network is a connected graph.

According to our knowledge, only the two following works [5, 18] take interest in the computation of the number of computation steps required by their algorithms. The algorithm in [5] has an upper bound on the number of steps of $O(\Delta.n^3)$ (Δ being the maximum node degree in the network). The algorithm in [5] is not silent, so some nodes change infinitely often their state. The silent algorithm in [18] has a convergence time $O(D^2)$ rounds having at most $O(n^6)$ computation steps. All these cited works assume that the network is a connected graph.

Self-stabilizing routing algorithm. In [19], Bein et al. present a self-stabilizing algorithm building local routing tables under the fair daemon (the tables ensure the routing from any node v to its t closest nodes) in $O(D)$ rounds in the connected component, but in $O(t)$ rounds within the disconnected component. Choosing the parameter t correctly helps to tackle the count-to-infinity problem. However, it means that in order to use their solution an upper bound on the network size has to be known. Cobb and Huang [20] propose an algorithm dedicated to the construction of shortest-path trees defined by any maximizable routing metrics. This algorithm works without any knowledge on the network but the correction is proved only for a restrictive class of asynchronous scenarios, namely the centralized weakly-fair daemon. Unfortunately, no upper bound on the number of execution steps is given.

Leader election algorithms. Another way to tackle our problem is to focus on the problem of leader election, as in [21, 22]. It has the advantage of running under the most general daemon, the unfair one, without any knowledge about the network topology. In [21], for each component, a BFS tree rooted at the selected leader is built within $4n + 11D + 4$ rounds. Whereas in [22], leader election is performed within $3n + D$ rounds by building a spanning tree rooted at the leader. In the case of routing, it is obviously preferable to have a shortest-path tree instead of a spanning tree, therefore a realistic candidate for our purpose would be [21]. Note that D stands for the diameter of the unweighted network.

These algorithms build, in each component, a tree rooted at the node with the smallest identifier. They could be tweaked to get semi-uniform algorithms that perform disconnection detection at the same time. To do so, instead of electing nodes based on their identifier, election would be done based on shared variables (and lexicographical order). Let us call these variables id, and change the algorithm in such a way that every node maintains its variable id equal to its identifier preceded by 0 if its r, or a 1 otherwise. After running this slightly modified election algorithm, every node can detect whether it belongs to the connected component of r by simply observing the first character of the elected node's identifier. This modification would only add a single round to the convergence time. However, it is not clear what would be the convergence time of these two algorithms for weighted networks. Moreover it is an interesting question to investigate whether it is possible to build a shortest-path tree in a smaller number of rounds.

Disjunction algorithm. Another potential candidate for our problem would be the algorithm proposed in [23] to solve the disjunction problem. It is silent and works with the assumption of an unfair daemon. This algorithm has a convergence time of $O(n)$ rounds but unfortunately no precise bound on the step complexity is given.

As a conclusion, there are no self-stabilizing algorithms in the literature that both maintain shortest-path trees and detect disconnected components in the most general scheduling scenario, the *unfair* daemon, while providing bounds on the number of rounds and the number of execution steps. Table 1 shows a summary of the BFS and shortest-path tree algorithms dealing with disconnected components.

1.2. Model

A distributed system S is an undirected graph $G = (V, E)$ where the vertex set V is the set of nodes and the edge set E is the set of communication links. A link $\{u, v\}$ belongs

	[19]	[20]	[21] with a simple modification (cf. Sec. 1.1)	[23]	this paper
Initial knowledge	param. $t \geq D$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
Handle weights	yes	yes	no	no	yes
Time (rounds)	$O(t)$	$O(D * deg)$	$4n + 11D + 4$	$O(n)$	$2n + D$
Distributed daemon	yes	no	yes	yes	yes
Fairness	weakly-fair	weakly-fair	unfair	unfair	unfair

Table 1: Related works on BFS and shortest-path tree algorithms dealing with disconnected components.

to E if and only if u and v can directly communicate (links are bidirectional); so, u and v are neighbors. We note by $\Gamma(v)$ the set of v's neighbors: $\Gamma(v) = \{u \in V \mid \{u, v\} \in E\}$. Each edge $\{u, v\}$ has a positive weight denoted $w(u, v)$; this notion is naturally extended to paths: the weight of a path is the sum of its edge weights. In the following, D stands for the hop-diameter of the underlying graph, that is the maximum over all pairs (u, v) of the minimum number of edges in a shortest path from u to v. The weighted distance between the nodes u and v is denoted $d(u, v)$, it is the minimal weight of a path from u to v ($+\infty$ if no such paths exist).

Each node v maintains a set of shared variables such that v can read its own variables and those of its neighbors, but it can modify only its variables. The *state* of a node is defined by the values of its local variables. The union of states of all nodes determines the *configuration* of the system. The *program* of each node is a set of *rules*. Each rule has two parts, the guard and the action. The *guard* of a v's rule is a Boolean expression involving the state of the node v, and those of its neighbors. The *action* of a v's rule updates v's state. So, every rule will be graphically described by two braces. The first brace contains the predicates such that their conjunction is the rule guard; and the second brace contains the rule action (i.e. one or several local variable updates).

In a configuration, a rule can be executed only if it is *enabled*, i.e., its guard evaluates to true. A node is *enabled* in a given configuration if at least one of its rules is enabled. A configuration is said to be *terminal* if and only if no node is enabled. In a semi-uniform algorithm, all nodes except one, denoted r, perform the same distributed algorithm. The set V_r denotes the connected component of the distinguished node r. In anonymous networks, nodes do not have distinct identifiers. However, we assume that a node can distinguish its neighbors since out-links of every node can be locally numbered.

During a *computation step under the daemon* S, $c_i \rightarrow^S c_{i+1}$, one or several enabled nodes in configuration c_i are selected by the daemon S. Theses nodes will simultaneously and atomically read their neighbors states and then perform their actions so that the system reaches the configuration c_{i+1} from c_i. An *execution* e *under daemon* S is a sequence of configurations $e = c_0, c_1, \cdots$, where c_{i+1} is reached from c_i by one computation step under S: $\forall i \geqslant 0, c_i \rightarrow^S c_{i+1}$. The *centralized daemon* selects at each computation step only one node while a *distributed daemon* selects any non-empty set of enabled nodes. Any daemon that only produces weakly-fair executions, that is executions in which an always enabled node is eventually activated, is called a *weakly-fair daemon*. There is no requirement on the *unfair daemon*, it may produce non weakly-fair execution. Finally, we say that an execution e is *maximal* if it is infinite, or if it reaches a terminal configuration.

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

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

Stabilization time. We use the *round* notion to measure the time complexity. The first round of an execution $e = c_1, c_2, \cdots$ is the minimal prefix $e_1 = c_1, \cdots, c_j$, such that every node having an enabled rule in c_1 either executes a rule or is neutralized during a computation step of e_1. A node v is *neutralized* during a computation step $c_i \rightarrow c_{i+1}$, if v is enabled in c_i but not anymore in configuration c_{i+1}.

Let e' be the suffix of e such that $e = e_1 e'$. The second round of e is the first round of e', and so on.

The stabilization time is the number of rounds of an execution reaching a legitimate configuration from any initial one.

Definition 2 (Round of a component). The end of the $i+1$-st round in the (connected) component $H \subseteq G$ in a computation e is defined recursively as the configuration of the execution e where every node $v \in H(V)$ that was enabled at the end of the i-th round of e in H have been either activated or neutralized once.

Note that the first round of a component $H \subseteq G$ may be shorter than the (global) first round (when the component is not explicitly given, then the round is assumed to be global, that is for the whole graph). Indeed, some nodes outside of H may be enabled at the beginning of the execution and may take a long time before being activated or neutralized. More generally, the i-th round of a component may end earlier than the (global) i-th round.

Definition 3 (Node convergence). A node v is said to have converged to its final state s under the daemon S at the configuration c_1 if along all executions under S from c_1, the node v keeps its state s.

In this paper, we consider a particular problem called the *Disconnected Components Detection and rooted Shortest-Path tree Maintenance* (DCDSPM) problem. See Fig. 1 for an illustration.

Definition 4 (The DCDSPM problem). Given a not necessarily connected network with a distinguished node r, the DCDSPM problem consists in converging to a configuration where every node in the connected component of r knows an incident edge on a shortest-path leading to r, and any other node detects that it does not belong to the same connected component as r.

1.3. Our contribution

We present an algorithm solving the DCDSPM problem and called FDcD. It shares some ideas with the self-stabilizing silent shortest-path tree algorithm originally presented in [20] under the *centralized weakly-fair daemon*. The part of our algorithm common with

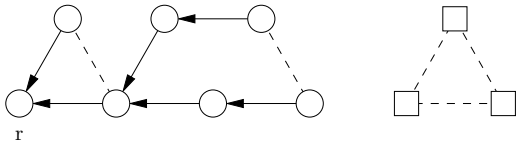


Figure 1: A network where the DCDSPM has been solved. Square nodes are the nodes claiming to be in a different connected component than r.

the algorithm in [20] is the use of a specific status (E/dirty), which is propagated from a node that has detected an anomaly to all nodes in its sub-tree. This technique is a well-known strategy to destroy illegal branches that was already used, for example, in [24] and [25]. The technical differences with [20] are detailed at the end of Section 2.

Our main contribution is an in-depth analysis of FDcD that holds under the most general daemon, the distributed unfair one. More precisely, we show that Algorithm FDcD fully solves the DCDSPM problem. That is, we prove that, on anonymous semi-uniform weighted networks, it builds a shortest-path tree rooted at r in V_r and explicitly isolates the nodes in all other connected components. No other algorithm solving the DCDSPM problem under the unfair daemon was known prior to this work. Note that our algorithm can easily be adapted to a wider class of routing metrics as shown in [20]. We also prove that Algorithm FDcD converges to a legitimate configuration within less than $2n + D$ rounds in any n-node weighted graph of hop-diameter D. Thus the time complexity of algorithm FDcD improves the one of [21] by a factor at least 2 in connected networks. Finally, we present an exponential lower bound on the worst-case number of steps, which also applies to the algorithm presented in [20]. According to our knowledge, this is the first time that a lower bound on the number of computation steps is provided for an algorithm building and maintaining a rooted shortest-path tree. The technique used to get this lower bound is novel and could be used to get lower bounds for other algorithms.

2. Algorithm FDcD

This section is devoted to the presentation of our algorithm, FDcD (Fast Disconnection Detection). The value of variable st indicates the status of the node: I for isolated (the node has no parent and no children), E for erroneous, and C for correct.

A non-isolated node u ($st_u \neq I$) has two other meaningful variables: the variable d_u containing the shortest weighted distance to r, and the variable $parent_u$ containing a pointer to the first out-link on the shortest path to r. Thus, only non-isolated nodes can belong to a branch (i.e. have children and/or a parent).

More formally, a node v is considered to be the *child* of another node u if v is a neighbor of u, neither u nor v has status I, v points to u ($parent_v = u$), and the known distance to the root via u ($d_u + \omega(u, v)$) is not larger than d_v. This leads to the following definition of the set $children_u$.

Definition 5 (Children of node u).
$children_u = \{v \in \Gamma(u) \mid (st_u \neq I) \wedge (st_v \neq I) \wedge (parent_v = u) \wedge (d_v \geq d_u + \omega(u, v))\}$

$$R_r \begin{cases} \begin{cases} P_{\text{root}}(u) \equiv (st_r \neq C) \vee (parent_r \neq r) \vee (d_r \neq 0) \end{cases} \\ \begin{cases} st_r \leftarrow C \\ parent_r \leftarrow r \\ d_r \leftarrow 0 \end{cases} \end{cases}$$

Figure 2: Algorithm FDcD on node r.

For example, in Fig. 5.a, p_1 has the isolated status I, so $children_{p_1} = \emptyset$. Besides, no node u satisfies $(d_u > d_{p_2}) \wedge (parent_u = p_2))$ so p_2 has no children: $children_{p_2} = \emptyset$. Finally, $children_{p_3} = \{p_4\}$ because p_4 is the only node u that satisfies $(d_u > dp_3) \wedge (parent_u = p_3)$.

We are now ready to provide the full description of the algorithm. The single rule for node r is given in Figure 2. The rules for the other nodes are given in Figure 3.

$$R_C \begin{cases} \begin{cases} P_{\text{create}}(u) \equiv (st_u \neq C) \wedge (children_u = \emptyset) \wedge (\exists v \in \Gamma(u) \mid st_v = C) \\ P_{\text{update}}(u) \equiv (\exists v \in \Gamma(u) \mid (st_v = C) \wedge (d_v + \omega(u,v) < d_u)) \\ P_{\text{correct}}(u) \equiv [\,(parent_u \notin \Gamma(u)) \vee (d_u \neq d_{parent_u} + \omega(u, parent_u)) \\ \qquad \vee (st_{parent_u} \neq C) \vee (st_u \neq C)\,] \\ \qquad \wedge [\exists v \in \Gamma(u) \mid (st_v = C) \wedge (d_v + \omega(u,v) = d_u)\,] \end{cases} \\ \begin{cases} st_u \leftarrow C \\ parent_u \leftarrow \text{argmin}_{(v \in \Gamma(u)) \wedge (st_v = C)}(d_v + \omega(u,v)) \\ d_u \leftarrow d_{parent_u} + \omega(u, parent_u) \end{cases} \end{cases}$$

$$R_E \begin{cases} \begin{cases} P_{\text{error}}(u) \equiv (st_u = C) \wedge (\forall v \in \Gamma(u) \mid (d_u < d_v + \omega(v,u)) \vee (st_v \neq C)) \end{cases} \\ \begin{cases} st_u \leftarrow E \end{cases} \end{cases}$$

$$R_I \begin{cases} \begin{cases} P_{\text{isolate}}(u) \equiv (st_u = E) \wedge (children_u = \emptyset) \wedge (\forall v \in \Gamma(u) \mid st_v \neq C) \end{cases} \\ \begin{cases} st_u \leftarrow I \end{cases} \end{cases}$$

Figure 3: Algorithm FDcD on node u.

Only nodes with status C may gain new children; and only nodes without children and with the status C may increase the value of their variable d (rule R_C). These two properties ensure that the execution of the rule R_C by a node u does not create any anomaly (because a node u doing R_C during a computation step has no children and it cannot gain children during this step). A given node u detects an anomaly in the

relationship with its parent in four cases:

- the parent node is not in its neighborhood — $(parent_u \notin \Gamma(u))$ is satisfied;
- the value of d_u is not coherent with the value of d_{parent_u} — $(d_u \neq d_{parent_u} + \omega(u, parent_u))$ is satisfied;
- it, or its parent, has not status C — $(st_u \neq C) \vee (st_{parent_u} \neq C)$ is satisfied;
- it is not the best out-link for the destination r — $P_{\text{update}}(u)$ is satisfied.

Besides, a node v is said to be an alternative parent for node u if it has status C and if:

- v is a better out-link than $parent_u$ (i.e. the cost of the path from u to r going through v is smaller than the cost of the path going through $parent_u$) — $P_{\text{update}}(u)$ is satisfied;
- or v is an as good out-link as $parent_u$ — $(st_v = C) \wedge (d_v + \omega(u, v) = d_u)$ is satisfied.

Note that the main difference with the algorithm presented in [20] is in the definition of an alternative parent. In their definition, a node is not considered to be an alternative parent in the case described by the second item.

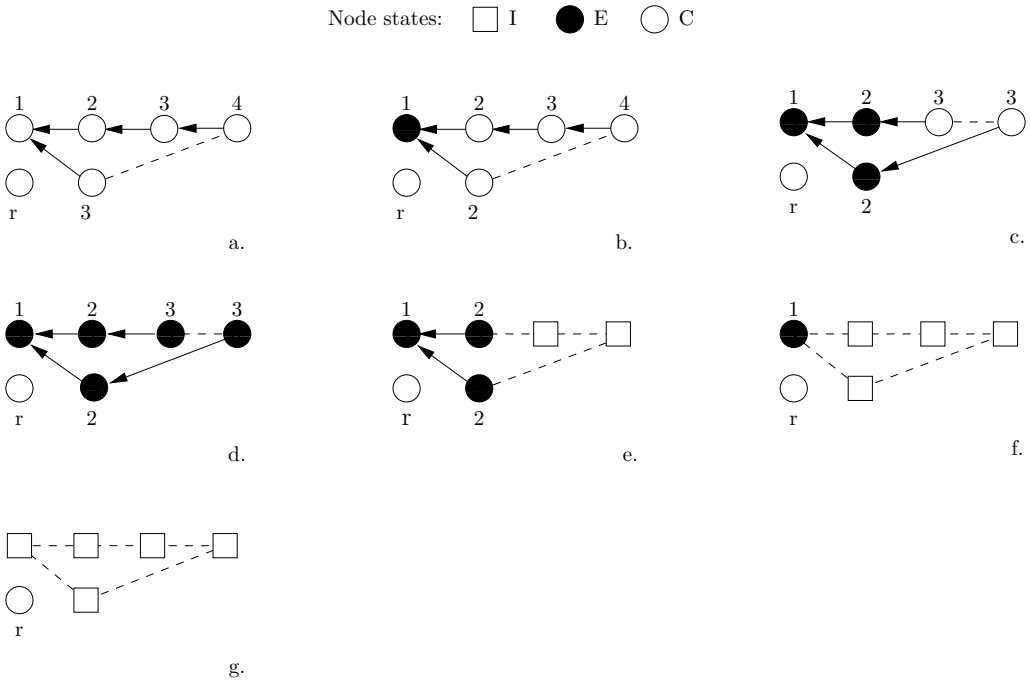


Figure 4: A synchronous execution

If a correct node u (i.e. with $st(u) = C$) detects an anomaly in the relationship with its parent, then u is enabled because $P_{\text{update}}(u)$, $P_{\text{correct}}(u)$, or $P_{\text{error}}(u)$ is satisfied. More precisely, $P_{\text{update}}(u)$ or $P_{\text{correct}}(u)$ is satisfied when u has an alternative parent; $P_{\text{error}}(u)$ is satisfied when u does not have an alternative parent.

When a node u detects an anomaly in the relationship with its parent and there is no alternative parent, u takes the status E (rule R_E). Then, u's children have an anomaly in the relationship with their parent, u. Thus the nodes in the sub-tree rooted at u will take the status E or will change sub-tree. When a leaf has the error status it quits its branch: either it becomes isolated (rule R_I) or it joins a "correct" branch (rule R_C). Therefore every erroneous sub-tree is eventually deleted.

A synchronous execution is presented in Fig. 4 (i.e. in each computation step of the execution, all enabled nodes perform their enabled action). The graph has two disconnected components: one containing only r and another one containing the other nodes. During the first step, a node performs the rule R_C to decrease the value of its d variable. During the second step, another node performs the rule R_C to change branch in the illegal tree. Meanwhile, during the first three computation steps, the E status is propagated from the illegal root to the leaves by the execution of the rule R_E. More precisely, during the first step, the illegal root performs this rule; during the second step, the children of the illegal root perform this rule; finally during the third step, the leaves of the illegal tree perform this rule. Then, during the last three computation steps, the I status is propagated backward from the leaves to the illegal root (nodes execute the rule R_I).

Any configuration during the execution of algorithm FDcD induces a BFS tree rooted at node r that spans a subset of V_r, a forest rooted at different *illegal roots* and some isolated nodes.

Definition 6 (Correct state). A node u is said to be in a correct state if: $st_u = C$, $d_u = d(u, r)$ and, if u is not the root r, $parent_u \in \Gamma_u$ and $d(parent_u, r) = d_u - \omega(u, parent_u)$.

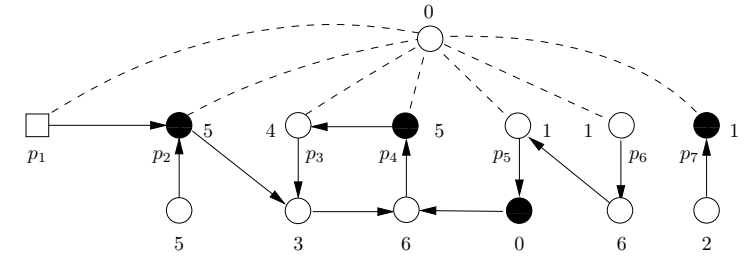
In Fig. 5.a, nodes on the first line (i.e. p_1 to p_7) are neighbors of the root, which is in its correct state. Nodes p_1 and p_2 satisfy P_{create} because they do not have the correct status and they are childless. Nodes p_3 and p_4 can decrease their d value and thus satisfy P_{update}. The distance value of p_5, p_6, and p_7 is equal to the final correct value 1, so they do not satisfy P_{update}. However, $parent_{p_5}$ and p_7 have the erroneous status, which implies that $P_{\text{correct}}(p_5)$ and $P_{\text{correct}}(p_7)$ are satisfied. Finally, the value of d_{p_6} is not coherent with the value of $d_{parent_{p_6}}$; so $P_{\text{correct}}(p_6)$ is satisfied.
After the computation step where all nodes on the first line perform the rule R_c, the configuration shown in Fig. 5.b is reached: all nodes on the first line are in their correct state.

Definition 7 (Legitimate state). A node u is said to be in a legitimate state if:

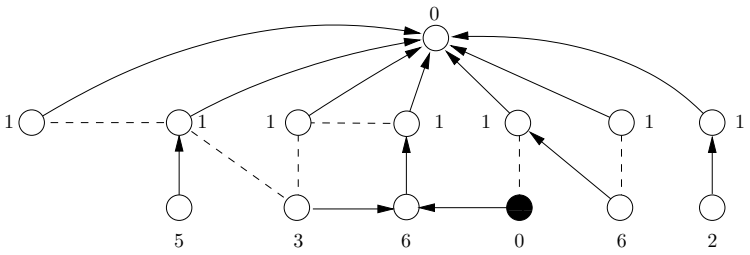
- it belongs to V_r and is in a correct state;
- or it does not belong to V_r and it has status I.

Definition 8 (Legitimate configuration). A legitimate configuration is a configuration where every node is in a legitimate state.

Coming back to the algorithm of [20], we point out some differences. First of all, the condition for a node to set its status to E/dirty is stronger in our algorithm than in [20]. In [20], as soon as a node has detected an anomaly or its parent has E/dirty



a. Configuration before the executions of the rules R_C



b. Configuration after the executions of the rules R_C

Figure 5: An illustration of several R_C executions.

status, then the node can take the E/dirty status. In our algorithm, only a node that cannot find an alternative parent has to take the E/dirty status, if it has detected an anomaly or if its parent has the E/dirty status. Another difference is that the rules of a node are not exclusive in [20]. It means that a node can have several enabled rules at the same time. The algorithm of [20] is thus not deterministic (even without considering the non-determinism coming from the asynchrony). There is also no discussions about the behavior of the presented algorithm in disconnected graphs and the proofs only concern the case of the fair daemon.

3. Correctness and convergence time of algorithm FDcD

We start this section by proving that the set of terminal configurations coincides with the set of legitimate configurations. This will be done thanks to the following two lemmas, the first one dealing with the connected components that do not contain node r, if some exist, and the second one dealing with the connected component V_r containing root node r.

3.1. Correctness

Lemma 1. *For any connected component H not containing node r, any terminal configuration in H is a legitimate configuration.*

PROOF. The proof is done by contradiction. So consider that for some connected component H not containing node r, there exists a terminal configuration in which at least one node has not status I.

Further assume that there exists some node that has status C. Consider a node $u \in H$ with status C having the smallest distance value d_u. By construction, u can apply rule R_E, which is in contradiction with the configuration being terminal. Therefore any node that does not have status I must have status E.

Consider now a node $u \in H$ that has status E having the largest distance value d_u. By construction and from the previous point, this node has no child and no neighbor have status C. Therefore node u can apply rule R_I, and we obtain again a contradiction, which concludes the proof of the lemma. □

Lemma 2. *Any terminal configuration within the connected component V_r is legitimate.*

PROOF. The proof is done by contradiction. Let consider it exists some non-legitimate terminal configuration of the connected component V_r.

Further, assume that there exists some node that has status E. Consider a node u of V_r with status E having the largest distance value d_u. Note that no node v that has status C can be a child of u, otherwise v could apply rule R_E or rule R_C. Therefore, node u has no child and thus can apply rule R_I or rule R_C, a contradiction.

Nodes have thus either status C or I. Assume now that there exists some node that has status I. Consider some node u with status I having at least one neighbor with status C. Such a neighbor node must exist because we are considering a connected component without any node with status E, but with at least one node that has status C, namely node r. Obviously, node u can apply rule R_C, a contradiction. So every node in V_r must have status C.

Now consider a node u in V_r having the smallest distance value d_u among the nodes in V_r that are not in a correct state. Then, either it exists some node v with status C in $\Gamma(u)$ such that $d_u \geq d_v + \omega(u, v)$, or not. If such a node v exists then node u can apply rule R_C. If it does not, then, by definition, it can apply rule R_E. In both those cases there is a contradiction, which concludes the proof. □

After noticing that any legitimate configuration is a terminal one, we conclude with the following corollary.

Corollary 1. *The set of terminal configurations coincide with the set of legitimate configurations.*

3.2. Convergence

We now prove that algorithm FDcD always terminates within $2n + D - 2$ rounds under a weakly-fair daemon, where D is the hop-diameter of the connected component containing r. Before proceeding with the proof, let us introduce some useful concepts.

Definition 9 (Branch). A *branch* is a maximal sequence of nodes $v_1, \cdots, v_k$, for some integer $k \geq 1$, such that none of the nodes have status I and, for every $i \leq k$, we have $v_i \in children_{v_{i+1}}$. The node v_i is said to be at *depth* $k - i$. If $v_k = r$ but the state of r is not terminal, or simply if $v_k \neq r$, the branch is said to be *illegal*, otherwise, the branch is said to be *legal*.

The first lemma essentially claims that all nodes that are in illegal branches progressively switch to status E within n rounds, in order of increasing depth (illegal branches may vary during the process though).

Lemma 3. *Fix any integer $i \geq 1$, and any connected component H. Starting from the beginning of round i of H, there does not exist any node of H both in state C and at depth less than $i-1$ in an illegal branch.*

PROOF. We prove this lemma by induction on i. The base case $i = 1$ is obvious so assume that the lemma holds for some integer $i \geq 1$. Consider any node u of H both with status C and depth $i-1$ in an illegal branch at the beginning of round i of H. If $u = r$, then r executes R_r during round i of H. Otherwise ($u \neq r$), by induction hypothesis, the parent of u is not in state C at that time. Therefore u is enabled at the beginning of round i. During round i, it will either execute rule R_E and thus switch to state E, or it will execute rule R_C (making it switch branch).

Note that, from the beginning of round i, no node can ever choose a parent which is at depth smaller than $i-1$ in an illegal branch because those nodes will never be in state C, by induction hypothesis. This is also true for node u if it applies rule R_C in round i. Therefore, no node can become in state C at depth smaller than i in an illegal branch. This concludes the proof of the lemma. □

Root node r does not belong to an illegal branch after the first round. Therefore, after the first round, the number of nodes of an illegal branch cannot be more than $n-1$. We thus obtain the following corollary.

Corollary 2. *For any connected component H, once round $n-1$ in H has terminated, no node in an illegal branch in H has status C.*

The next lemma essentially claims that, within at most $n-1$ subsequent rounds, the maximal length of an illegal branch progressively decreases until no illegal branches remain.

Lemma 4. *Fix any integer $i \geq 0$, and any connected component H. Starting from the beginning of round $n+i$ in H, there does not exist any node of H at depth larger or equal to $n-i-1$ in an illegal branch.*

PROOF. We prove the lemma by induction on i. The base case $i = 0$ is obvious so assume that the lemma holds for some integer $i \geq 0$. By induction hypothesis, at the beginning of round $n+i$, no node is at depth larger or equal to $n-i-1$. Therefore, the nodes at depth $n-i-2$ in an illegal branch have no children and are thus enabled at the beginning of round $n+i$. These nodes will thus all be executed within round $n+i$ (they cannot be neutralized as no children can connect to them). We conclude the proof by noticing that, from Corollary 2, once round $n-1$ has terminated, every node in an illegal tree is in state E, and thus any node in an illegal branch that gets executed from this time will not be anymore in any illegal branch. □

Corollary 3. *For any connected component H, once round $2n-2$ in H has terminated, there are no illegal branches in H.*

Note that in a connected component that does not contain the root r, there are no legal branches. Since the only way for a node to be in no branch is to have status I, we obtain the following result.

Corollary 4. *For any connected component H not containing r, after $2n-2$ rounds in H, every node v of H has status I.*

After $2n-2$ rounds, the connected components not containing r are in a legitimate state. In the connected component V_r containing r, Algorithm FDcD may need additional rounds so that the correct distances to r are correctly propagated.

In the following lemma, we use the notion of hop-distance to r defined below.

Definition 10 (Hop-distance to the root node r). A node v is said to *be at k hops from r* if k is the minimum number of edges of a shortest path from v to r.

Lemma 5. *Consider any integer $i \geq 0$. For any execution of Algorithm FDcD, starting from the beginning of round $2n-2+i$, every node in component V_r at most i hops from r is in a correct state.*

Proof. Let us prove the lemma by induction on i. Firstly, we need to remark that after one single round, node r has necessary converged to the correct state. So the base case $i = 0$ holds, as we can assume n to be at least 2. Secondly, at round $2n-2$, from Corollary 3, every node either belongs to a legal branch or have status I, thus any node $v \in V_r$ always stores a distance d such that $d \geq d(v,r)$, its actual weighted distance to r. By induction hypothesis, every node at at most i hops from r has converged to a correct state before round $2n+i-1$. Therefore, at the beginning of round $2n+i-1$, every node v at $i+1$ hops from r which is not in a correct state has rule R_C enabled. Thus, at the end of round $2n+i-1$, every node at at most $i+1$ hops from r is in a correct state (such nodes cannot be neutralized during this round). Also, these nodes will never change their state since there are no nodes other than their parent that can make them get closer to r than their current parent. □

Putting together all the results of this section, we obtain, for algorithm FDcD, the following theorem.

Theorem 1. *Under a weakly-fair daemon, Algorithm FDcD always converges to a legitimate state within $2n+D-2$ rounds, where D is the hop-diameter of the connected component V_r containing node r.*

4. Convergence under an unfair daemon

In this section, we will prove that algorithm FDcD always converges to a legitimate state, even under an unfair daemon. The proof, by contradiction, will go as follows. After noticing that a node activated infinitely often must execute rule R_C infinitely many times, we will prove that nodes activated infinitely often must have globally increasing distance values. This means that these nodes will eventually behave as if the nodes activated a finite number of times do not exist. This will lead to a contradiction, as we proved before that a connected component has to become silent after a finite number of rounds.

Lemma 6. *If at some time a node has been executed k times, then it must have executed rule R_C at least $\lfloor \frac{k-2}{3} \rfloor$ many times.*

PROOF. When a node with status E is enabled, it can either execute rule R_C or rule R_I. Moreover, a node with status I can only execute rule R_C. Thus between two consecutive executions of rule R_C by a node, only two other rule executions can happen. □

Let us now introduce a useful notation for the next lemmas.

Definition 11. A node u is said to execute a rule with (distance) value δ if the distance value d_u is equal to δ immediately after this rule execution.

Lemma 7. *Rule R_C cannot be executed infinitely often with the same distance value.*

PROOF. For the purpose of contradiction, consider any (infinite) execution e of algorithm FDcD in which rule R_C is applied infinitely often with the same distance value. Let $d_{\min}$ be the minimum such infinitely often used value. Let v be some node applying infinitely often rule R_C with distance value $d_{\min}$. Now consider some suffix e' of e in which no node with a distance value smaller than $d_{\min}$ will ever apply any rule. Note that such a suffix e' must exist, by definition of $d_{\min}$.

Let consider the maximal suffix e'' of e' starting when node v has a parent u such that $d_u = d_{\min} - \omega(u, v)$. By definition of e', node u will remain in state C and be the better possible parent within e'', therefore node v will not apply any rule in e'', contradicting the assumption that node v applies infinitely often rule R_C. □

We are now ready to conclude about the convergence under an unfair daemon.

Lemma 8. *Every execution is finite.*

PROOF. For the purpose of contradiction, let us assume that there exists an infinite execution e. Let F, resp. $\overline{F}$, be the set of nodes executed finitely, resp. infinitely, many times in this execution, and let F' be the set of nodes in F that are neighbors of at least one node in $\overline{F}$. Note that the set F is necessarily non-empty as it contains at least node r.

Let execution e_1 be a suffix of e in which every node $v \in F$ is never executed. In e_1, only the nodes from $\overline{F}$ will be executed. Let $d_{\max}$ be the maximum distance stored in d_v for any node $v \in F$ within e_1. From Lemma 7, if a node executes an infinite number of steps during an execution of algorithm FDcD, then it will necessarily change its distance an infinite number of times. Moreover, distances stored at a given node cannot be negative.

Thus, there exists a suffix e_2 of e_1 such that for any node $\overline{v}$ in $\overline{F}$, $d_{\overline{v}} > d_{\max} + \omega_v$, where ω_v is the maximum weight of an edge incident to $\overline{v}$.

Within e_2, a node $v' \in F'$ cannot have status C, otherwise any node $\overline{v}$ that belongs to $\Gamma(v') \cap \overline{F}$ would apply R_C with distance value at most $d_{\max} + \omega(\overline{v}, v')$ which would be in contradiction with the definition of e_2. Moreover, we have $d_{\overline{v}} > d_{v'}$, and thus v' does not belong to $children_{\overline{v}}$.

Looking at the algorithm, one can observe that, if a rule can be applied for a node $v \in \overline{F}$ during e_2, then it can still be applied after removing the nodes in F' from the graph.

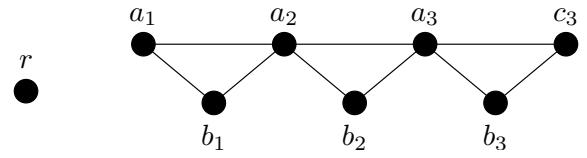


Figure 6: The graph $\mathcal{G}_3$

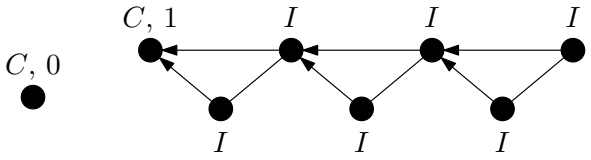


Figure 7: The initial configuration $init_3$

In other words, the nodes in $\overline{F}$ can have the same execution in the graph obtained after removing the nodes in F. Now consider any connected component H of $\overline{F}$. Notice that r is not a node of H. Since all nodes in H are activated infinitely many times, it means that there are an infinite number of rounds in H, without the nodes reaching a terminal configuration in H. Corollary 4 establishes that every node of H are isolated after at most $2n-2$ rounds in H. Then, every node of H are and stays disabled forever, they have reached their terminal state. This concludes the proof of this lemma. □

A corollary of this lemma is that any execution is weakly-fair because there are no nodes that are always enabled (at the end of a finite execution, no nodes are enabled). Therefore, the bound on the number of rounds from Theorem 1 also applies in the case of an unfair daemon. We obtain the following main theorem.

Theorem 2. *Under an unfair daemon, Algorithm* FDcD *always converges to a legitimate state within a finite number of steps and in at most* $2n + D - 2$ *rounds, where* D *is the hop-diameter of the connected component* V_r *containing node* r*.*

5. Lower bound on the number of steps

In this section, we prove that the step complexity of our algorithm can be as large as $2^{n/2}$ in some n-node unweighted graphs. This lower bound is based on a family of graphs $\{\mathcal{G}_k\}_{k\geq 1}$, defined as follows.

Definition 12 (Graph $\mathcal{G}_k$). For any positive integer k, the graph $\mathcal{G}_k$ consists of one isolated node r, and k triangles with node sets $\{a_1, b_1, c_1\}, \cdots, \{a_k, b_k, c_k\}$, where c_i is merged with a_{i+1}, for $1 \leq i < k$. (See Fig. 6.)

In order to obtain the desired number of steps, we consider a particular initial configuration for each graph $\mathcal{G}_k$, called $init_k$. In this configuration, the root node r is correctly initialized: $st_r = C$, $parent_r = r$, and $d_r = 0$. Moreover, we have $st_{a_1} = C$, $parent_{a_1} = a_1$, and $d_{a_1} = 1$. Finally, all other nodes are in state I. As an illustrative example, the initial configuration of the graph $\mathcal{G}_3$ is presented in Fig. 7.

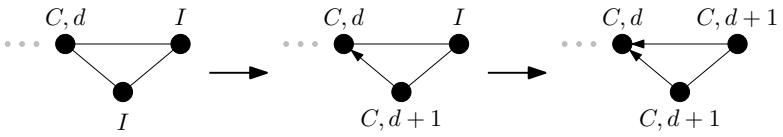


Figure 8: Rule applications performed by b_k and c_k after each application of Rule R_C by a_k.

The intuition behind the exponential lower bound is the following. Node a_1 cannot stay in state C and thus eventually switches to state E, and finally to state I when becoming childless. As for node c_1, it first switches to sate C, having a_1 as parent, then it gains state E when a_1 switches to state E. Node c_1 eventually comes back in state C when becoming childless, but this time choosing b_1 as its parent. The state E then propagates to c_1 via b_1, both of them eventually switching to state I when becoming childless. To summarize, c_1 is basically doing twice the transition from C to E when a_1 is doing it once. More generally, we will prove that the number of such transitions by node c_{i+1} is twice the number of such transitions by node c_i, yielding to the desired bound.

In order to derive more formally our lower bound on the number of steps, we prove by induction the following property.

Definition 13 (Property $\mathcal{P}_k$). Given a positive integer k, we say that Property $\mathcal{P}_k$ holds if and only if there exists an execution of our algorithm on $\mathcal{G}_k$, starting from the configuration $init_k$, such that the node c_i applies exactly 2^{i-1} times the sequence of rules R_C, R_E, R_C, R_E, R_I, for $1 \leq i \leq k$.

Lemma 9. *For any positive integer k, Property $\mathcal{P}_k$ holds.*

Proof. We prove this lemma by induction on k.

Let us first consider the graph $\mathcal{G}_1$, in the initial configuration $init_1$. The execution of our algorithm proving Property $\mathcal{P}_1$ is defined by the following rule executions: R_C by b_1, R_C by c_1, R_E by a_1, R_E by c_1, R_C by c_1, R_E by b_1, R_E by c_1, and finally R_I by c_1. This execution is depicted by the concatenation of Fig. 8 and 9.

Fix any positive integer k and assume that Property $\mathcal{P}_k$ holds. We will prove that Property $\mathcal{P}_{k+1}$ holds as well. Let $\mathcal{E}_k$ be the desired execution on $\mathcal{G}_k$ whose existence is proved by the induction hypothesis. We now extend this execution to fit it to the graph $\mathcal{G}_{k+1}$, with the initial configuration $init_{k+1}$, as follows.

As long as any node other than c_k is concerned in $\mathcal{E}_k$, we can apply the same computation steps in $\mathcal{G}_{k+1}$. Besides, whenever a rule R_C is applied by c_k in $\mathcal{E}_k$, we ensure that b_{k+1} and c_{k+1} (the neighbors that c_k has in $\mathcal{G}_{k+1}$ but not in $\mathcal{G}_k$) are in state I and thus do not modify the satisfiability of any predicate from rule R_C (it is the case for the first execution of R_C by c_k and it will be the case by construction for the other executions). Moreover, after each application of R_C by c_k, the correct status is propagated to the the nodes b_{k+1} and c_{k+1} by having them apply rule R_C, see Fig. 8.

Similarly, no predicates from R_E see their satisfiability changed by the existence of these two nodes when a rule R_E is applied by c_k in $\mathcal{E}_k$. Indeed, by construction, both b_{k+1} and c_{k+1} store a greater distance than d, the distance stored by c_k, because they have just applied the rule R_C based on the value d, and thus have distance value $d + 1$.

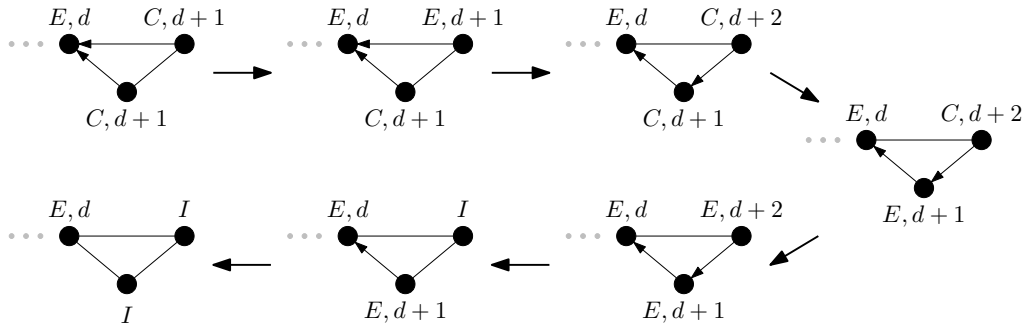


Figure 9: Rule applications performed by b_k and c_k after each application of Rule R_E by a_k.

Moreover, after each such application of R_E by c_k, we propagate twice the status E to c_{k+1} (the first time directly, then through b_{k+1}) thanks to the following rule executions: R_E by c_{k+1}, R_C by c_{k+1}, R_E by b_{k+1}, and R_E by c_{k+1}. Then, both these nodes switch back to their initial state I in order for the remainder of the execution $\mathcal{E}_k$ to continue (both nodes apply Rule R_I). The sequence of added rule applications is depicted in Fig. 9.

Finally, whenever a rule R_I is applied by c_k in $\mathcal{E}_k$, our construction ensures that b_{k+1} and c_{k+1} are in state I and thus do not modify the satisfiability of any predicate from rule R_I.

This extended execution is thus well defined and has the required properties, which concludes the proof of this lemma. □

This technical lemma allows us to obtain the main result of this section.

Corollary 5. *For any $n \geq 4$, there exists a n-node graph and a particular initial configuration from which our algorithm uses at least $2^{n/2}$ steps to stabilize.*

Proof. For even values of n, this follows directly from Lemma 9. For odd values of n, it is sufficient to just add an additional isolated node to the graph $\mathcal{G}_{(n-3)/2}$. □

Note that this construction can be used to obtain the same lower bound for the algorithm presented in [20].

6. Conclusion

Our algorithm is the first self-stabilizing algorithm to simultaneously compute shortest-path trees and discover disconnected components in the most general scheduling scenario, the *unfair* daemon. We also prove theoretically a linear upper bound on the number of rounds. Our work can be useful to save both memory space and messages in the setting of routing in dynamic networks without any restriction on the behavior of the distributed system.

This work, as many of the works on self-stabilization, considers the powerful model of shared memory with composite atomicity, in which, in a single atomic step, a node can read its own variables and those of its neighbors and can modify its own variables. This

model allows for simpler and more elegant algorithms and proofs than other more realistic models, but does not restrict the applicability of the algorithms designed in it. Indeed, there exist general self-stabilizing transformers between this model and more realistic ones. Dolev, Israeli, and Moran proposed in [16] a transformer from the composite atomicity model to the more realistic model of read/write atomicity: in one atomic step, a node can read the state of one of its neighboring nodes, or update its own state, but not both simultaneously. Another transformer [26] permits to go from this model to a message passing model with FIFO but unreliable bidirectional channels of communication. Finally, one can use a further transformer [27] to handle unreliable capacity-bounded non-FIFO channels.

Our contribution can lead to some new results: the algorithm can easily be adapted to a wider class of routing metrics as shown in [20] and our exponential lower bound also holds for the algorithm in [20].

The algorithm FDcD analyzed in this paper is very efficient in terms of number of rounds but might be costly in terms of number of steps. We have to point out that for self-stabilizing algorithms, very few results on the number of steps are known and the majority of analyses are done on restrictive scheduling scenarios.

Thus, the first natural open problem is to design a shortest-path algorithm with a polynomial step complexity while preserving a small round complexity under the *unfair daemon*, even if the disconnection discovering goal is removed. One has to keep in mind that under the unfair daemon, the combinatorial explosion on the number of state configurations can quickly lead to a large number of cases to be considered. There is some hope that getting a polynomial step complexity could be possible, but such an algorithm is unknown today.

It would also be interesting to experimentally compare this algorithm to (potentially) less robust ones such as [20, 21, 23] for which either the upper bound on the number of execution steps or the behavior under the most general daemon are unknown. In this paper, our focus goes to hard instances: we showed that our algorithm achieves stabilization for any initial state and any sequence of actions. Having experimental results would be very interesting but lies outside the scope of this paper, as it would require significant specific research advances. Indeed, on one hand, random uniform schedulings are very unlikely to behave as worst-case schedulings, and devising realistic probabilistic distributions remains challenging and largely open. On the other hand, it would be interesting to have experimental results for instances in which the number of execution steps tends to be large. Nevertheless, we are not aware of any experimental technique that would permit to compare different algorithms under equivalently hard-case scenarios. It seems very hard to design such experimental routines and proposing one would be a very interesting result on its own. The main difficulty in designing such an experimental protocol is that both the initial states of the processes and the scheduling of hard- (or worst-) case scenarios strongly depend on the algorithm. Thus it is not trivial at all to find a general way of generating hard instances. We therefore believe that these questions are intriguing but challenging open research subjects related to our work.

7. Acknowledgements

This study was partially supported by the ANR project DESCARTES (ANR-16-CE40-0023) and by the ANR project ESTATE (ANR-16-CE25-0009-03).

This study has been carried out in the frame of "the Investments for the future" Programme IdEx Bordeaux – CPU (ANR-10-IDEX-03-02).

8. References

[1] C. Glacet, N. Hanusse, D. Ilcinkas, C. 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, 2014, pp. 120–134.

[2] A. Leon-Garcia, I. Widjaja, Communication Networks, 2nd Edition, McGraw-Hill, Inc., New York, NY, USA, 2004.

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

[4] C. Gavoille, C. Glacet, N. Hanusse, D. Ilcinkas, On the communication complexity of distributed name-independent routing schemes, in: the 27th International Symposium on Distributed Computing (DISC'13), Springer LNCS 8205, 2013, pp. 418–432.

[5] A. Cournier, S. Devismes, V. Villain, Light enabling snap-stabilization of fundamental protocols, ACM Transactions on Autonomous and Adaptive Systems 4 (1).

[6] S.-T. Huang, N.-S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, Information Processing Letters 41 (2) (1992) 109–117.

[7] C. Boulinier, F. Petit, V. Villain, When graph theory helps self-stabilization, in: Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing (PODC'04), 2004.

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

[9] T. C. Huang, J.-C. Lin, A self-stabilizing algorithm for the shortest path problem in a distributed system, Computers & Mathematics with Applications 43 (1) (2002) 103–109.

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

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

[12] A. Arora, M. Gouda, T. Herman, Composite routing protocols, in: the 2nd IEEE Symposium on Parallel and Distributed Processing (SPDP'90), 1990, pp. 70–78.

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

[14] C. Johnen, S. Tixeuil, Route preserving stabilization, in: the 6th International Symposium on Self-stabilizing System (SSS'03), Springer LNCS 2704, 2003, pp. 184–198.

[15] N. Chen, H. Yu, S. Huang, A self-stabilizing algorithm for constructing spanning trees, Information Processing Letters 39 (1991) 147–151.

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

[17] L. Blin, M. Potop-Butucaru, S. Rovedakis, 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, 2010, pp. 50–64.

[18] A. Cournier, S. Rovedakis, V. 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, 2011, pp. 159–174.

[19] D. Bein, A. K. Datta, V. Villain, Self-stabilizing local routing in ad hoc networks, The Computer Journal 50 (2) (2007) 197–203.

[20] J. Cobb, C.-T. Huang, Stabilization of maximal-metric routing without knowledge of network size, in: Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on, IEEE, 2009, pp. 306–311.

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

[22] K. Altisen, A. Cournier, S. Devismes, A. Durand, F. Petit, Self-stabilizing leader election in polynomial steps, in: the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'14), Springer LNCS 8736, 2014, pp. 106–119.

[23] A. K. Datta, S. Devismes, L. L. Larmore, Self-stabilizing silent disjunction in an anonymous network, in: 14th International Conference on Distributed Computing and Networking (ICDCN 2013), Springer LNCS 7730, 2013, pp. 148–160.

[24] S.-T. Huang, N.-S. Chen, Self-stabilizing depth-first token circulation on networks, Distributed Computing 7 (1) (1993) 61–66.
[25] C. Johnen, J. Beauquier, Distributed self-stabilizing depth-first token circulation with constant memory, in: Proceedings of the Second Workshop on Self-Stabilizing Systems (WSS'95), 1995, pp. 4.1–4.15.
[26] Y. Afek, G. M. Brown, Self-stabilization over unreliable communication media, Distributed Computing 7 (1) (1993) 27–34.
[27] S. Dolev, S. Dubois, M. Potop-Butucaru, S. Tixeuil, Stabilizing data-link over non-fifo channels with optimal fault-resilience, Inf. Process. Lett. 111 (18) (2011) 912–920.