Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Networks

Christian Glacet, Nicolas Hanusse, David Ilcinkas, Colette Johnen

▶ **To cite this version:**

Christian Glacet, Nicolas Hanusse, David Ilcinkas, Colette Johnen. Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Networks. Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2014), Sep 2014, Paderborn, Germany. pp.120 - 134, 10.1007/978-3-319-11764-5_9 . hal-01111188

HAL Id: hal-01111188
https://hal.science/hal-01111188

Submitted on 29 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Disconnected components detection and rooted shortest-path tree maintenance in networks*

Glacet Christian[1], Hanusse Nicolas[2], Ilcinkas David[2], and Johnen Colette[1]

[1] Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
[2] 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. Without any assumption on the asynchronous model (*unfair* daemon), with no knowledge of the network and within an anonymous network, we present a silent self-stabilizing algorithm solving this more demanding task and running in less than $2n + D$ rounds for a network of n nodes and hop-diameter D.

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 dynamicity 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 other node 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 [LGW04]: 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 fast as possible.

In practice, the most standard technics 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 not guaranteed without any assumption

*Partially supported by the ANR project DISPLEXITY (ANR-11-BS02-014). This study has been carried out in the frame of "the Investments for the future" Programme IdEx Bordeaux – CPU (ANR-10-IDEX-03-02).

(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 toward a legitimate configuration can be often provided by self-stabilizing algorithms. However, solutions that can be found in the literature are dedicated to the maintenance of a BFS tree or shortest paths, but only 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 [AGM$^+$08,GGHI13], 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 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 two algorithms for a single root for an *unfair daemon* but our solutions hold 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 algorithms work in anonymous networks in the semi-uniform model.

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, the difficulty is to find an algorithm running in the worst sequence of processes execution in an asynchronous setting. Models of processes execution are called *daemons*. In [CS94,HL02], 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 [Hua05b], Tetz Huang proves that the algorithms in [CS94,HL02] 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 [Hua05a].

In [AGH90,CG02,JT03], self-stabilizing algorithms for the single-destination shortest-path problem are presented; these algorithms ensure the loop-free property: after any edge cost changes, even during the re-building phase, there is always a path from any node to the destination. To sum up, 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 [CYH91] under the central daemon. Huang et al. present the first self-stabilizing BFS tree construction in [HC92] under the unfair distributed daemon. In [CYH91,HC92], the exact network size has to be known by all nodes. Dolev, Israeli and Moran in [DIM93] present the first self-stabilizing BFS spanning-tree construction algorithm under read/write atomicity.

Blin et al. in [BPBRT10] present an 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.

Self-stabilizing routing algorithm. In [BDV07], 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, in but $O(t)$ 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.

Leader election algorithms. Surprisingly, one way to get closer to our goal is to focus on the problem of leader election, as in [DLV11,ACD$^+$14], under the very general daemon, the unfair one, without any knowledge about the network topology. In [DLV11] (resp. in [ACD$^+$14]), for each component, a BFS tree rooted at the selected leader is built within $4n+11D+4$ rounds (resp. $3n+D$ rounds). Note that D stands for the diameter of the unweighted network.

Since, in each component, the selected leader is the node with smallest identifier, one could change a little bit these uniform algorithms into semi-uniform algorithms, by forcing the node r to have the smallest identifier in adding a single bit to every identifiers. However, this trick can work only for $k = 1$ and it is not clear what would be the convergence time for a weighted network.

1.2 Model

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

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).

A rule can be executed only if it is *enabled*, i.e., its guard evaluates to true. A node is *enabled* 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. V_r denotes the connected component of 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. The fair daemon may select several nodes at each step, but it produces only fair executions (an always enabled node is eventually activated). There is no requirement on the unfair daemon; so unfair executions are produced by the unfair daemon.

We say that an execution e is *maximal* if it is infinite, or if it reaches a terminal configuration. We note by $\mathcal{C}$ the set of all possible configurations, and by $\mathcal{E}^S$ the set of all maximal executions under the daemon S. The set of maximal executions under the daemon S starting from a particular configuration $c \in \mathcal{C}$ is denoted $\mathcal{E}_c^S$.

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.*

We can notice that the i-th round in a component $H \subseteq G$ can end earlier than the i-th round (when the component is not explicitly given then the round is global).

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.*

1.3 Our contributions

We present two self-stabilizing silent algorithms on anonymous semi-uniform weighted networks working under the unfair daemon. Both algorithms build a shortest-path tree rooted at r in V_r, and isolate the nodes in the other connected components.

We first present a simple distributed algorithm, namely Algorithm DcD, which is quite natural. We show that the convergence time may unfortunately be as high as $\Omega(n^2)$ rounds in some n-node graphs of large diameter.

Changing a little bit this algorithm, we end up with a second algorithm FDcD. This latter algorithm converges to a legitimate configuration within less than $2n + D$ rounds in any n-node weighted graph of hop-diameter D.

2 Our algorithms

This section is devoted to the presentation of our two algorithms, DcD (Disconnection Detection) and FDcD (Fast Disconnection Detection). These algorithms are using the same key idea and are thus very similar (although their performances are different).

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).

The single rule for node r is the same for both algorithms (Figure 1).

Definition 4 (Children of node u). $children_u =$
$\{v \in \Gamma(u) \mid (st_u \neq I) \wedge (st_v \neq I) \wedge (parent_v = u) \wedge (d_v \geq d_u + \omega\{u, v\})\}$

Definition 5 (Correct state). *A node u is said to be in a correct state if:*

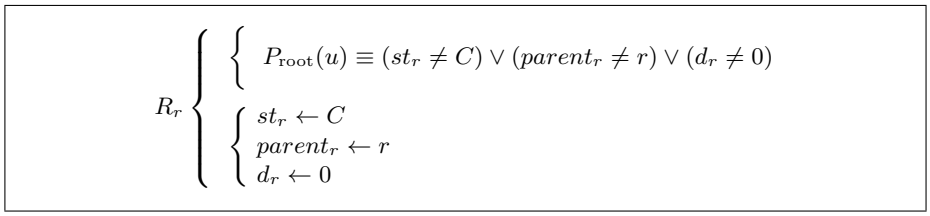

Fig. 1. Algorithm DcD or FDcD on node r.

- *its status variable is* C,
- *its distance variable is set to* $d(u, r)$ *the weighted distance from* u *to* r *and*
- *the weighted distance* $d(parent_u, r)$ *of* $parent_u$ *to* r *is* $d_u - \omega\{u, parent_u\}$.

Definition 6 (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 7 (Legitimate configuration). *A legitimate configuration is a configuration where every node is in a legitimate state.*

2.1 A first and simple algorithm : Algorithm DcD

Algorithm DcD is given in figure 2. It is roughly based on the following idea. Whenever a node detects a local anomaly, it somehow detaches from its parent, warns its whole sub-tree, and then reconnects to another tree.
A given node u detects an anomaly in the relationship with its parent in four cases:

- the parent node is not in its neighborhood;
- it is not the best out-link for the destination r;
- the value of d_u is not coherent with the value of d_{parent_u};
- it, or its parent, has not status C.

When a node u detects an anomaly in the relationship with its parent, it takes the status E (rule R_C). Notice that the error status is propagated in sub-trees. When a leaf has the error status, then it can quit its tree: either it becomes isolated (rule R_I) or it joins a "correct" branch (rule R_C). So any erroneous sub-trees are eventually deleted.

Only nodes with status C may gain new children; and only nodes without children may change the value of their variable d or *parent* (rule R_C) to join a new branch. These two properties ensure that the execution of the rule R_C by a node u does not create anomaly (because a node u doing R_C during a computation step has no children and it cannot gain children during this step).

We can show that algorithm DcD converges to a legitimate configuration. However, for graphs with large diameter ($D = \Theta(n)$) it may converge in $\Omega(n^2)$

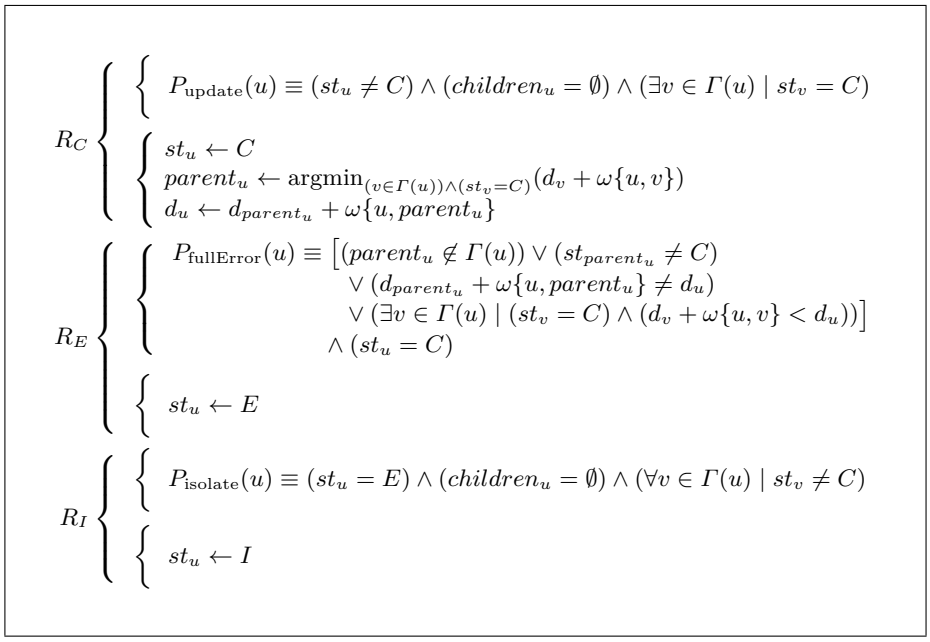

Fig. 2. Algorithm DcD on node u.

rounds. This lower bound is based on a graph $\mathcal{G}_n$ defined right after and presented in Figure 3. It uses several copies of the undermentioned graph H_i.

Definition 8 (Graph H). *The graph called H is a 5-node graph composed by a path (a, b, c, e) where node e and a are connected together via an intermediate node f.*

Definition 9 (Graph $\mathcal{G}_n$). *The graph $\mathcal{G}_n$ is composed of n copies of graph H: $H_0, H_1, \ldots, H_{n-1}$. To build $\mathcal{G}_n$ simply connect every H_i to H_{i+1} by merging nodes e_i and a_{i+1} (the index indicates the copy).*

This graph $\mathcal{G}_n$ has $4n + 1$ nodes and diameter $2n$.

Lemma 1. *For the graph $\mathcal{G}_n$ of $O(n)$ nodes, algorithm* DcD *may converge to a legitimate configuration within $\Omega(n^2)$ rounds.*

Proof. Let consider a graph $\mathcal{G}$ and set node a_0 to be the root of $\mathcal{G}_n$. In this proof we will mainly consider two possible configurations for any graph H_i:

- An *illegitimate configuration*, called *ic*, where nodes b_i, c_i, e_i, f_i have their parents variables respectively set to a_i, b_i, c_i, a_i, which leads node e_i to have its distance set to $d_{a_i} + 3$.
- And also the *legitimate configuration*, where nodes b_i, c_i, e_i, f_i have their parents variables respectively set to a_i, b_i, f_i, a_i and $d_{a_i} = 2i$. This leads node e_i to have its distance set to $2i + 2$.

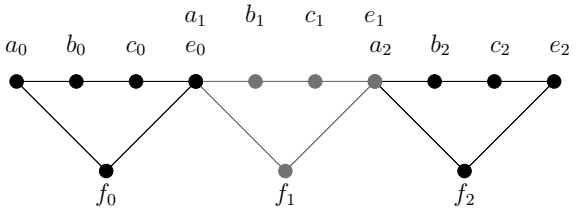

Fig. 3. $\mathcal{G}_3$ with the node names

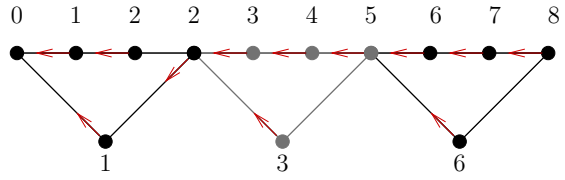

Fig. 4. Configuration X_1 of $\mathcal{G}_3$

First, we can notice that an illegitimate configuration can turn to a legitimate configuration for graph H_i only if node a_i already stores a distance of $2i$, which implies that every H_k such that $k < i$ is in legitimate state.

Let us define the configuration X_i for $0 \leq i \leq n$ as follow. The root node a_0 is in correct state, every H_k for $k < i$ is in the legitimate configuration and every H_j for $j \geq i$ is in the illegitimate configuration. The configuration X_1 is shown on figure 4 for $n = 3$. X_n is the legitimate configuration of $\mathcal{G}_n$.

We will study one execution from X_0 in which configurations X_i such that $i \in (1, 2, \ldots, n)$ are successively reached. This execution has n steps, during the i^{th} step, configuration X_i is reached from configuration X_{i-1}.

Let us compute the number of rounds required to execute the $(i+1)^{\text{th}}$ step. The node e_i has to switch its parent from c_i to f_i, resulting in changing its distance from $2i+3$ to $2i+2$. The difficulty is that to change its parent, node e_i must have the status I. The node e_i will get the status I only after the status E is propagated into e_i sub-tree which takes 3 rounds. After what, the status I is propagated from e_{n-1} to e_i which takes $3(n-i-1)$ more rounds. Now, we need to bring every H_j such that $j > i$ into the illegitimate configuration. That can be done by activating successively, for every H_j such that $j > i$, every node except node f_j, after what the remaining nodes $(f_j \mid j > i)$ can be activated in an arbitrary order. Therefore, the $(i+1)^{\text{th}}$ step takes at least $3(n-i-1)+3$ rounds.

In this execution, the total number of rounds needed to converge to the legitimate configuration on $\mathcal{G}_n$ is thus greater than $\sum_{i=1}^{n} 3(n-i)$. Which gives the lower bound of $\Omega(n^2)$. □

2.2 A more efficient solution

With algorithm FDcD, presented in Figure 5, a node u joins a correct branch sooner than with algorithm DcD. Nevertheless, no anomaly is created when a node modifies the value of its variable d or *parent*.

When an anomaly is detected by a node u in the relationship with its parent, if there is an alternative parent to connect to, then u changes parent (rule R_C). Otherwise, u takes status E (rule R_E). Algorithm FDcD's rules are quite similar to DcD's, the only differences lie in R_C and R_E guards. A node p is an alternative parent for node u if it has status C and if :

- p is a better out-link than $parent_u$ (i.e. the cost of the path from u to r going through p is smaller than the cost of the path going through $parent_u$);
- or d_u matches d_p (i.e. $d_p + \omega\{p, u\} = d_u$).

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.

$$
\begin{array}{l}
R_C \left\{ \begin{array}{l}
\left\{ \begin{array}{l}
P_{\text{create}}(u) \equiv (st_u \neq C) \wedge (children_u = \emptyset) \wedge (\exists v \in \Gamma(u) \mid st_v = C) \\
P_{\text{update}}(u) \equiv (\exists v \in \Gamma(u) \mid (st_v = C) \wedge (d_v + \omega\{u, v\} < d_u)) \\
P_{\text{correct}}(u) \equiv [\,(parent_u \notin \Gamma(u)) \vee (d_u \neq d_{parent_u} + \omega\{u, parent_u\}) \\
\qquad\qquad \vee (st_{parent_u} \neq C) \vee (st_u \neq C)\,] \\
\qquad\qquad \wedge [\exists v \in \Gamma(u) \mid (st_v = C) \wedge (d_v + \omega\{u, v\} = d_u)\,]
\end{array} \right. \\
\left\{ \begin{array}{l}
st_u \leftarrow C \\
parent_u \leftarrow \text{argmin}_{(v \in \Gamma(u)) \wedge (st_v = C)}(d_v + \omega\{u, v\}) \\
d_u \leftarrow d_{parent_u} + \omega\{u, parent_u\}
\end{array} \right.
\end{array} \right. \\
R_E \left\{ \begin{array}{l}
\left\{ \begin{array}{l}
P_{\text{error}}(u) \equiv (st_u = C) \wedge (\forall v \in \Gamma(u) \mid (d_u < d_v + \omega\{(v, u\}) \vee (st_v \neq C))
\end{array} \right. \\
\left\{ \begin{array}{l}
st_u \leftarrow E
\end{array} \right.
\end{array} \right. \\
R_I \left\{ \begin{array}{l}
\left\{ \begin{array}{l}
P_{\text{isolate}}(u) \equiv (st_u = E) \wedge (children_u = \emptyset) \wedge (\forall v \in \Gamma(u) \mid st_v \neq C)
\end{array} \right. \\
\left\{ \begin{array}{l}
st_u \leftarrow I
\end{array} \right.
\end{array} \right.
\end{array}
$$

Fig. 5. Algorithm FDcD on node u.

3 Correctness and convergence time of algorithm FDcD

All the proofs in this section hold for both algorithms, except Lemma 6 and thus Theorem 1, that hold only for 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.

Lemma 2. *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 the 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 the 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 3. *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 the 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 the 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.*

We now prove that algorithm FDcD always terminates within $2n + D - 2$ rounds under a 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 10 (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.

Lemma 4. *Fix any integer $i \geq 1$, and any connected component H. Starting from the beginning of round i in 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. If $u = r$, then r executes R_r. Otherwise, by induction hypothesis, the parent of u is not in state C. 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.

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. Therefore, no node can become in state C at depth smaller than i. This is also true for node u if it applies rule R_C in round i. 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 5. *Fix any integer $i \geq 0$, any of the two algorithms, 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 11 (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 6. *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. □

Note that algorithm DcD eventually reach an identical configuration where every node is either isolated or belongs to the tree rooted at r. But from this particular configuration algorithm DcD takes more time to converge to a legitimate state. We know from Lemma 1 that the convergence to a legitimate state can take at least $\Omega(n^2)$ additional rounds for some settings of graphs and configurations.

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

Theorem 1. *Under a 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. Note that all the lemmas in this section also hold for Algorithm DcD.

Lemma 7. *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 12. *A node* u *is said to execute a rule with (distance) value* dist *if the distance value* d_u *is equal to* dist *immediately after this rule execution.*

Lemma 8. *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 9. *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 8, if a node executes an infinite number of steps during an execution of algorithm FDcD, then it will necessary 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. 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 $\overline{F}$. 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. This is in contradiction with Corollary 4, and this concludes the proof of this lemma. □

Summarizing the results proved so far, 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.

References

[ACD+14] Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing leader election in polynomial steps. Technical Report hal-00980798 (http://hal.archives-ouvertes.fr/hal-00980798), VERIMAG, MIS, LIP6, INRIA Rocquencourt, April 2014.

[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.

[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.

[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.

[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.

[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.

[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.

[DLV11] 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.

[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.

[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.

[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.

[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.

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