# A New Self-Stabilizing Minimum Spanning Tree Construction with Loop-free Property

Lélia Blin, Maria Gradinariu Potop-Butucaru, Stephane Rovedakis, Sébastien Tixeuil

# A New Self-Stabilizing Minimum Spanning Tree Construction with Loop-free Property

Lélia Blin[1,2] Maria Potop-Butucaru[2,3] Stéphane Rovedakis[1]
Sébastien Tixeuil[2,4]

May 14, 2009

**Abstract**

The minimum spanning tree (MST) construction is a classical problem in Distributed Computing for creating a globally minimized structure distributedly. Self-stabilization is versatile technique for forward recovery that permits to handle any kind of transient faults in a unified manner. The loop-free property provides interesting safety assurance in dynamic networks where edge-cost changes during operation of the protocol.

We present a new self-stabilizing MST protocol that improves on previous known approaches in several ways. First, it makes fewer system hypotheses as the size of the network (or an upper bound on the size) need *not* be known to the participants. Second, it is loop-free in the sense that it guarantees that a spanning tree structure is always preserved while edge costs change dynamically and the protocol adjusts to a new MST. Finally, time complexity matches the best known results, while space complexity results show that this protocol is the most efficient to date.

[1]Université d'Evry, IBISC, CNRS, France.
[2]Univ. Pierre & Marie Curie - Paris 6, LIP6-CNRS UMR 7606, France.
[3]INRIA REGAL, France,
[4]INRIA Futurs, Project-team Grand Large.

# 1 Introduction

Since its introduction in a centralized context [24, 21], the minimum spanning tree (or MST) construction problem gained a benchmark status in distributed computing thanks to the influential seminal work of [12]. Given an edge-weighted graph $G = (V, E, w)$, where $w$ denotes the edge-weight function, the MST problem consists in computing a tree $T$ spanning $V$, such that $T$ has minimum weight among all spanning trees of $G$.

One of the most versatile technique to ensure forward recovery of distributed systems is that of *self-stabilization* [5, 6]. A distributed algorithm is self-stabilizing if after faults and attacks hit the system and place it in some arbitrary global state, the system recovers from this catastrophic situation without external (*e.g.* human) intervention in finite time. A recent trend in self-stabilizing research is to complement the self-stabilizing abilities of a distributed algorithm with some additional *safety* properties that are guaranteed when the permanent and intermittent failures that hit the system satisfy some conditions. In addition to being self-stabilizing, a protocol could thus also tolerate a limited number of topology changes [8], crash faults [14, 1], nap faults [9, 22], Byzantine faults [10, 2], and sustained edge cost changes [3, 19].

This last property is specially relevant when building spanning trees in dynamic networks, since the cost of a particular edge is likely to evolve through time. If a MST protocol is *only* self-stabilizing, it may adjust to the new costs in such a way that a previously constructed MST evolves into a disconnected or a looping structure (of course, in the abscence of new edge cost changes, the self-stabilization property guarantees that *eventually* a new MST is constructed). Of course, if edge costs change unexpectedly and continuously, a MST can not be maintained at all times. Now, a packet routing algorithm is *loop free* [13, 11] if at any point in time the routing tables are free of loops, despite possible modification of the edge-weights in the graph (*i.e.*, for any two nodes $u$ and $v$, the actual routing tables determines a simple path from $u$ to $v$, at any time). The *loop-free* property [3, 19] in self-stabilization guarantees that, a spanning tree being constructed (not necessarily a MST), then the self-stabilizing convergence to a "minimal" (for some metric) spanning tree maintains a spanning tree at all times (obviously, this spanning tree is not "minimal" at all times). The consequence of this safety property in addition to that of self-stabiization is that the spanning tree structure can still be used (e.g. for routing) while the protocol is adjusting, and makes it suitable for networks that undergo such very frequent dynamic changes.

**Related works** Gupta and Srimani [17] have presented the first self-stabilizing algorithm for the MST problem. It applies on graphs whose nodes have unique identifiers, whose edges have integer edge weights, and a weight can appear at most once in the whole network. To construct the (unique) MST, every node performs the same algorithm. The MST construction is based on the computation of all the shortest paths (for a certain cost function) between all the pairs of nodes. While executing the algorithm, every node stores the cost of all paths from it to all the other nodes. To implement this algorithm, the authors assume that every node knows the number $n$ of nodes in the network, and that the identifiers of the nodes are in $\{1, \ldots, n\}$. Every node $u$ stores the weight of the edge $e_{u,v}$ placed in the MST for each node $v \neq u$. Therefore the algorithm requires $\Omega(\sum_{v \neq u} \log w(e_{u,v}))$ bits of memory at node $u$. Since all the weights are distinct integers, the memory requirement at each node is $\Omega(n \log n)$ bits.

Higham and Lyan [18] have proposed another self-stabilizing algorithm for the MST problem. As [17], their work applies to undirected connected graphs with unique integer edge weights and unique node identifiers, where every node has an upper bound on the number of nodes in the system. The algorithm performs roughly as follows: every edge aims at deciding whether it eventually belongs to the MST or not. For this purpose, every non tree-edge $e$ floods the

| | metric | size known | unique weights | memory usage | loop-free |
|---|---|---|---|---|---|
| [17] | **MST** | yes | yes | $\Theta(n \log n)$ | no |
| [18] | **MST** | upper bound | yes | $\Theta(n \log n)$ | no |
| [3] | SP | upper bound | **no** | $\mathbf{\Theta(\log n)}$ | **yes** |
| [19] | SP | **no** | **no** | $\mathbf{\Theta(\log n)}$ | **yes** |
| This paper | **MST** | **no** | **no** | $\mathbf{\Theta(\log n)}$ | **yes** |

Table 1: Distributed Self-Stabilizing algorithms for the MST and loop-free SP problems

network to find a potential cycle, and when $e$ receives its own message back along a cycle, it uses information collected by this message (*i.e.*, the maximum edge weight of the traversed cycle) to decide whether $e$ could potentially be in the MST or not. If the edge $e$ has not received its message back after the time-out interval, it decides to become tree edge. The core memory of each node holds only $O(\log n)$ bits, but the information exchanged between neighboring nodes is of size $O(n \log n)$ bits, thus only slightly improving that of [17].

To our knowledge, *none* of the self-stabilizing MST construction protocols is loop-free. Since the aforementioned two protocols also make use of the knowledge of the global number of nodes in the system, and assume that no two edge costs can be equal, these extra hypoteses make them suitable for static networks only.

Relatively few works investigate merging self-stabilization and loop free routing, with the notable exception of [3, 19]. While [3] still requires that a upper bound on the network diameter is known to every participant, no such assumption is made in [19]. Also, both protocols use only a reasonable amount of memory ($O(\log n)$ bits per node). However, the metrics that are considered in [3, 19] are derivative of the shortest path (*a.k.a.* SP) metric, that is considered a much easier task in the distributed setting than that of the MST, since the associated metric is *locally optimizable* [16], allowing essentially locally greedy approaches to perform well. By contrast, some sort of *global optimization* is needed for MST, which often drives higher complexity costs and thus less flexibility in dynamic networks.

**Our contributions** We describe a new self-stabilizing algorithm for the MST problem. Contrary to previous self-stabilizing MST protocols, our algorithm does not make any assumption about the network size (including upper bounds) or the unicity of the edge weights. Moreover, our solution improves on the memory space usage since each participant needs only $O(\log n)$ bits, and node identifiers are not needed.

In addition to improving over system hypotheses and complexity, our algorithm provides additional safety properties to self-stabilization, as it is loop-free. Compared to previous protocols that are both self-stabilizing and loop-free, our protocol is the first to consider non-monotonous tree metrics.

The key techniques that are used in our scheme include fast construction of a spanning tree, that is continuously improved by means of a pre-order construction over the nodes. The cycles that are considered over time are precisely those obtained by adding one edge to the evolving spanning tree. Considering solely that type of cycles reduces the memory requirement at each node compared to [17, 18] because the latter consider all possible paths connecting pairs of nodes. Moreover, constructing and using a pre-order on the nodes allows our algorithm to proceed in a completely asynchronous manner, and without any information about the size of the network, as opposed to [17, 18]. The main characteristics of our solution are presented in Table 1, where a boldface denotes the most useful (or efficient) feature for a particular criterium.

## 2 Model and notations

We consider an undirected weighted connected network $G = (V, E, w)$ where $V$ is the set of nodes, $E$ is the set of edges and $w : E \rightarrow \mathbb{R}^+$ is a positive cost function. Nodes represent processors and edges represent bidirectional communication links. Additionally, we consider that $G = (V, E, w)$ is a network in which the weight of the communication links may change value. We consider anonymous networks (i.e., the processor have no IDs), with one distinguished node, called the *root*[1]. Throughout the paper, the root is denoted $r$. We denote by $\deg(v)$ the number of $v$'s neighbors in $G$. The $\deg(v)$ edges incident to any node $v$ are labeled from 1 to $\deg(v)$, so that a processor can distinguish the different edges incident to a node.

The processors asynchronously execute their programs consisting of a set of variables and a finite set of rules. The variables are part of the shared register which is used to communicate with the neighbors. A processor can read and write its own registers and can read the shared registers of its neighbors. Each processor executes a program consisting of a sequence of guarded rules. Each *rule* contains a *guard* (boolean expression over the variables of a node and its neighborhood) and an *action* (update of the node variables only). Any rule whose guard is *true* is said to be *enabled*. A node with one or more enabled rules is said to be *privileged* and may make a *move* executing the action corresponding to the chosen enabled rule.

A *local state* of a node is the value of the local variables of the node and the state of its program counter. A *configuration* of the system $G = (V, E)$ is the cross product of the local states of all nodes in the system. The transition from a configuration to the next one is produced by the execution of an action at a node. A *computation* of the system is defined as a *weakly fair, maximal* sequence of configurations, $e = (c_0, c_1, \ldots c_i, \ldots)$, where each configuration $c_{i+1}$ follows from $c_i$ by the execution of a single action of at least one node. During an execution step, one or more processors execute an action and a processor may take at most one action. *Weak fairness* of the sequence means that if any action in $G$ is continuously enabled along the sequence, it is eventually chosen for execution. *Maximality* means that the sequence is either infinite, or it is finite and no action of $G$ is enabled in the final global state.

In the sequel we consider the system can start in any configuration. That is, the local state of a node can be corrupted. Note that we don't make any assumption on the bound of corrupted nodes. In the worst case all the nodes in the system may start in a corrupted configuration. In order to tackle these faults we use self-stabilization techniques.

**Definition 1 (self-stabilization)** *Let $\mathcal{L}_{\mathcal{A}}$ be a non-empty* legitimacy predicate[2] *of an algorithm $\mathcal{A}$ with respect to a specification predicate Spec such that every configuration satisfying $\mathcal{L}_{\mathcal{A}}$ satisfies Spec. Algorithm $\mathcal{A}$ is* self-stabilizing *with respect to Spec iff the following two conditions hold:*
*(i) Every computation of $\mathcal{A}$ starting from a configuration satisfying $\mathcal{L}_{\mathcal{A}}$ preserves $\mathcal{L}_{\mathcal{A}}$ (*closure*).*
*(ii) Every computation of $\mathcal{A}$ starting from an arbitrary configuration contains a configuration that satisfies $\mathcal{L}_{\mathcal{A}}$ (*convergence*).*

We define bellow a *loop-free* configuration of a system as a configuration which contains paths with no cycle between any couple of nodes in the system.

[1]Observe that the two self-stabilizing MST algorithms mentioned in the Previous Work section assume that the nodes have distinct IDs with no distinguished nodes. Nevertheless, if the nodes have distinct IDs then it is possible to elect one node as a leader in a self-stabilizing manner. Conversely, if there exists one distinguished node in an anonymous network, then it is possible to assign distinct IDs to the nodes in a self-stabilizing manner [7]. Note that it is not possible to compute deterministically a MST in a fully anonymous network (i.e., without any distinguished node), as proved in [17].

[2]A legitimacy predicate is defined over the configurations of a system and is an indicator of its correct behavior.

**Definition 2 (Loop-Free Configuration)** *Let $Cycle(u, v)$ be the following predicate defined for two nodes $u, v$ on configuration $C$, with $P(u, v)$ a path from $u$ to $v$ described by $C$:*

$$Cycle(u, v) \equiv \exists P(u, v), P(v, u) : P(u, v) \cap P(v, u) = \emptyset.$$

*A loop-free configuration is a configuration of the system which satisifes $\forall u, v : Cycle(u, v) = false$.*

We use the definition of a loop-free configuration to define a *loop-free stabilizing* system.

**Definition 3 (Loop-Free Stabilization)** *A distributed system is called loop-free stabilizing if and only if it is self-stabilizing and there exists a non-empty set of configurations such that the following conditions hold: (i) Every execution starting from a loop-free configuration reaches a loop-free configuration (closure). (ii) Every execution starting from an arbitrary configuration contains a loop-free configuration (convergence).*

In the sequel we study the loop-free self-stabilizing LoopFreeMSTproblem. The legitimacy predicate $\mathcal{L}_{\mathcal{A}}$ for the LoopFreeMSTproblem is the conjunction of the following two predicates: (i) a tree $T$ spanning the network is constructed. (ii) $T$ is a minimum spanning tree of $G$ (i.e., $\forall T', W(T) \leq W(T')$, with $T'$ be a spanning tree of $G$ and $W(S) = \sum_{e \in S} w(e)$ be the cost of the subgraph $S$).

# 3 The Algorithm LoopFreeMST

In this section, we describe our self-stabilizing algorithm for the MST problem. We call this algorithm LoopFreeMST. In the next section, we shall prove the correctness of this algorithm, and demonstrate that it satisfies all the desired properties listed in Section 1, including the loop-freedomness property. Let us begin by an informal description of LoopFreeMST aiming at underlining its main features.

## 3.1 High level description

LoopFreeMST is based on the red rule. That is, for constructing a MST, the algorithm successively deletes the edges of maximum weight within every cycle. For this purpose, a spanning tree is maintained, together with a pre-order labeling of its nodes. Given the current spanning tree $T$ maintained by our algorithm, every edge $e$ of the graph that is not in the spanning tree creates an unique cycle in the graph when added to $T$. This cycle is called *fundamental cycle*, and is denoted by $C_e$. (Formally, this cycle depends on $T$; Nevertheless no confusion should arise from omitting $T$ in the notation of $C_e$). If $w(e)$ is not the maximum weight of all the edges in $C_e$, then, according to the red rule, our algorithm swaps $e$ with the edge $f$ of $C_e$ with maximum weight . This swapping procedure is called an *improvement*. A straightforward consequence of the red rule is that if no improvements are possible then the current spanning tree is a minimum one.

Algorithm LoopFreeMST can be decomposed in three procedures:

- Tree construction
- Token label circulation
- Cycle improvement

The latter procedure (Cycle improvement) is in fact the core of our contribution. Indeed, the two first procedures are simple modifications of existing self-stabilizing algorithms, one for building a spanning tree, and the other for labelling its nodes. We will show how to compose the original procedure "Cycle improvement" with these two existing procedures. Note that "Cycle improvement" differs from the previous self-stabilizing implementation of the improvement swapping in [18] by the fact that it does not require any a priori knowledge of the network, and it is loop-free.

LoopFreeMST starts by constructing a spanning tree of the graph, using the self-stabilizing loop-free algorithm "Tree construction" described in [20]. The two other procedures are performed concurrently. A token circulates along the edges of the current spanning tree, in a self-stabilizing manner. This token circulation uses algorithms proposed in [4, 23] as follows. A non-tree-edge can belong to at most one fundamental cycle, but a tree-edge can belong to several fundamental cycles. Therefore, to avoid simultaneous possibly conflicting improvements, our algorithm considers the cycles in order. For this purpose, the token labels the nodes of the current tree in a DFS order (pre-order). This labeling is then used to find the unique path between two nodes in the spanning tree in a distributed manner, and enables computing the fundamental cycle resulting from adding one edge to the current spanning tree.

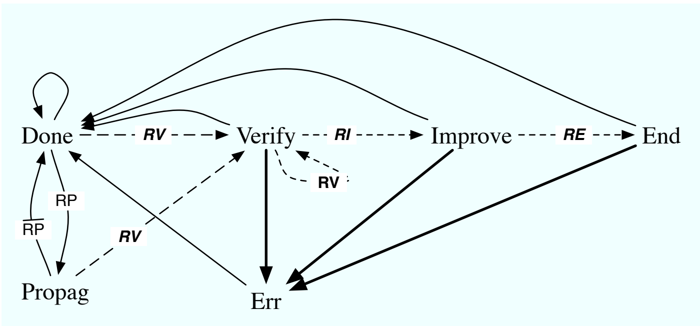


Figure 1: Evolution of the node's state in cycle improvement module. Rule $\mathsf{R}_{\mathsf{D}}$ is depicted in plain. Rule $\mathsf{R}_{\mathsf{Err}}$ is depicted in bold.

We now sketch the description of the procedure "Cycle improvement" (see Figure 1). When the token arrives at a node $u$ in a state `Done`, it checks whether $u$ has some incident edges not in the current spanning tree $T$ connecting $u$ with some other node $v$ with smaller label. If it is the case, then enters state `Verify`. Let $e = \{u, v\}$. Node $u$ then initiates a traversal of the fundamental cycle $C_e$ for finding the edge $f$ with maximum weight in this cycle. If $w(f) = w(e)$ then no improvement is performed. Else an improvement is possible, and $u$ enters State `Improve`. Exchanging $e$ and $f$ in $T$ results in a new tree $T'$. The key issue here is to perform this exchange in a loop-free manner. Indeed, one cannot be sure that two modifications of the current tree (i.e., removing $f$ from $T$, and adding $e$ to $T$) that are applied at two distant nodes will occur simultaneously. And if they do not occur simultaneously, then there will a time interval during which the nodes will not be connected by a spanning tree. Our solution for preserving loop-freedomless relies on a sequence of successive local and atomic changes, involving a single variable. This variable is a pointer to the current parent of a node in the current spanning tree. To get the flavor of our method, let us consider the example depicted on Figure 2. In this example, our algorithm has to exchange the edge $e = \{10, 12\}$ of weight 9, with the edge $f = \{7, 8\}$ of weight 10 (Figure 2(a)). Currently, the token is at node 12. The improvement is performed in two steps, by a sequence of two local changes. First, node 10 switches its parent from 8 to 12 (Figure 2(b)). Next, node 8 switches its parent from 7 to 10 (Figure 2(c)). A spanning tree is preserved at any time during the execution of these changes.

Note that any modification of the spanning tree makes the current labeling globally inaccurate, i.e., it is not necessarily a pre-order anymore. However, the labeling remains a pre-order in the portion of the tree involved in the exchange. For instance, consider again the example depicted on Figure 2(c). When the token will eventually reach node $A$, it will label it by some label $\ell > 12$. The exchange of $e = \{10, 12\}$ and $f = \{7, 8\}$ has not changed the pre-order for the fundamental cycle including edge $\{A, 12\}$. However, when the token will eventually reach node $B$ and label it $\ell' > \ell$, the exchange of $e = \{10, 12\}$ and $f = \{7, 8\}$ has changed the pre-order for the fundamental cycle including edge $\{B, 9\}$: the parent of node labeled 10 is labeled 12 whereas it should have a label smaller than 10 in a pre-order. When the pre-order is modified by an exchange, the inaccurately labeled node changes its state to `Err`, and stops the traversal of the fundamental cycle. The token is then informed that it can discard this cycle, and carry on the traversal of the tree.

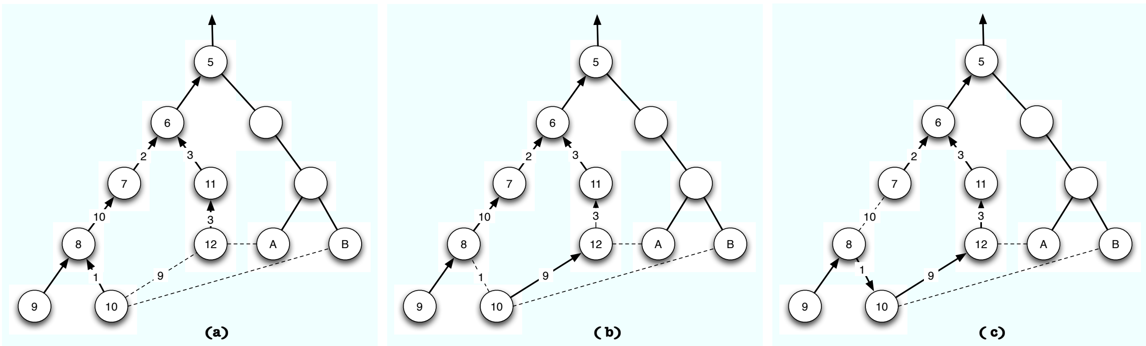


Figure 2: Example of a loop-free improvement of the current spanning tree. The direction of the edges indicate the parent relation. Edges in the spanning tree are depicted as plain lines; Edges not in the spanning tree are denoted by dotted lines.

### 3.2 Detailed level description

We now enter into the details of Algorithm LoopFreeMST. First, let us state all variables used by the algorithm. Later on, we will describe its predicates and its rules.

**Variables** For any node $v \in V(G)$, we denote by $N(v)$ the set of all neighbors of $v$ in $G$. Algorithm LoopFreeMST maintains the set $N(v)$ at every node $v$. We use the following notations:

- $\mathsf{parent}_v$: the parent of $v$ in the current spanning tree;
- $\mathsf{label}_v$: the integer label assigned to $v$;
- $\mathsf{d}_v$: the distance (in hops) from $v$ to the root in the current spanning tree;
- $\mathsf{state}_v$: the state of node $v$, with values in {`Done`, `Verify`, `Improve`, `End`, `Propag`, `Err`};
- $\mathsf{DefCycle}_v$: the pair of labels of the two extremities of the non tree-edge corresponding to the current fundamental cycle.
- $\mathsf{VarCycle}_v$: a pair of variables: the first one is the maximum edge-weight in the current fundamental cycle; the second one is a (boolean) variable in {Before, After};
- $\mathsf{suc}_v$: the successor of $v$ in the current fundamental cycle.

**Consistency rules** The first task executed by LoopFreeMST is to check the consistency of the variables of each node; See Figure 1. `Done` is the standard state of a node when this node has not the token, or is not currently visited by the traversal of a fundamental cycle. When the variables of a node are detected to be not coherent, the state of the node becomes `Err` thanks to rule $\mathsf{R_{Err}}$. There is one predicate in $\mathsf{R_{Err}}$ for each state, except for state `Propag`, to check whether the variables of the node are consistent (see Figure 3). The rule $\mathsf{R_D}$ allows the node to return to the standard state `Done`. More precisely, rule $\mathsf{R_D}$ resets the variables, and stops the participation of the node to any improvement.

$\mathsf{R_{Err}}$: **(Bad label)**
**If** $\mathsf{CoherentCycle}(v) \wedge \mathsf{Error}(v) \wedge \mathsf{DefCycle}[0]_v \neq \mathsf{label}_v \wedge \mathsf{EndPropag}(v)$ **then** $\mathsf{state}_v := \mathtt{Err}$;

$\mathsf{R_D}$: **(Improvement consistency)**
**If** $\neg\mathsf{CoherentCycle}(v) \wedge \mathsf{EndPropag}(v)$
**then** $\mathsf{state}_v := \mathtt{Done}; \mathsf{DefCycle}_v := (\mathsf{label}_v, \mathsf{done}); \mathsf{VarCycle}_v := (0, \mathsf{Before}); \mathsf{suc}_v := \emptyset$;

$\mathsf{CoherentCycle}(v) \equiv \mathsf{Coherent\_Done}(v) \vee \mathsf{Coherent\_Verify}(v) \vee \mathsf{Coherent\_Improve}(v) \vee \mathsf{Coherent\_End}(v) \vee \mathsf{Coherent\_Error}(v)$

$\mathsf{Coherent\_Done}(v) \equiv \mathsf{state}_v = \mathtt{Done} \wedge \mathsf{suc}_v = \emptyset \wedge \mathsf{DefCycle}_v = (\mathsf{label}_v, \mathsf{done}) \wedge \mathsf{VarCycle}_v = (0, \mathsf{Before})$

$\mathsf{Coherent\_Verify}(v) \equiv \mathsf{state}_v = \mathtt{Verify} \wedge \mathsf{suc}_v = \mathsf{Succ}(v) \wedge [(\mathsf{Init}(v) \wedge \mathsf{VarCycle}_x = (0, \mathsf{Before})) \vee \mathsf{Nds\_Verify}(v)]$

$\mathsf{Coherent\_Improve}(v) \equiv \mathsf{state}_v = \mathtt{Improve} \wedge \mathsf{suc}_v = \mathsf{Succ}(v) \wedge \mathsf{DefCycle}_v = \mathsf{DefCycle}_{\mathsf{parent}_v} \wedge \mathsf{VarCycle}_v = \mathsf{VarCycle}_{\mathsf{parent}_v}$

$\mathsf{Coherent\_End}(v) \equiv \mathsf{state}_v = \mathtt{End} \wedge \mathsf{DefCycle}_v = \mathsf{DefCycle}_{\mathsf{parent}_v} \wedge (\mathsf{NdDel}(v) \vee \mathsf{Ask\_EI}(v))$

$\mathsf{Coherent\_Error}(v) \equiv \mathsf{state}_v = \mathtt{Err} \wedge (\mathsf{suc}_v = \mathsf{Succ}(v) = \emptyset \vee \mathsf{Ask\_E}(v)) \wedge \mathsf{DefCycle}_v = \mathsf{DefCycle}_{\mathsf{Pred}(v)}$

$\mathsf{CoherentTree}(v) \equiv (v = r \wedge \mathsf{d}_v = 0 \wedge st_v = N) \vee (v \neq r \wedge Safe_v \wedge rw_v = \mathsf{d}_v) \vee \mathsf{state}_{\mathsf{parent}_v} = \mathtt{Improve} \vee \mathsf{state}_{\mathsf{parent}_v} = \mathtt{Propag}$

$\mathsf{Ask\_V}(v) \equiv \mathsf{state}_{\mathsf{Pred}(v)} = \mathtt{Verify}$

$\mathsf{Ask\_I}(v) \equiv (\mathsf{state}_{\mathsf{Pred}(v)} = \mathtt{Improve} \wedge \mathsf{VarCycle}[1]_{\mathsf{Pred}(v)} = \mathsf{Before}) \vee (\mathsf{state}_{\mathsf{suc}_v} = \mathtt{Improve} \wedge \mathsf{VarCycle}[1]_{\mathsf{suc}_v} = \mathsf{After})$

$\mathsf{Ask\_EI}(v) \equiv (\exists u \in N(v), \mathsf{parent}_u = v \wedge \mathsf{state}_u = \mathtt{End} \wedge \mathsf{DefCycle}_u = \mathsf{DefCycle}_v)$

$\mathsf{Ask\_E}(v) \equiv \mathsf{suc}_v \neq \emptyset \wedge \mathsf{state}_{\mathsf{suc}_v} = \mathtt{Err} \wedge \mathsf{DefCycle}_v = \mathsf{DefCycle}_{\mathsf{suc}_v}$

Figure 3: Corrections predicates used by LoopFreeMST.

$\mathsf{Tree\_Edge}(v, u) \equiv \mathsf{parent}_v = u \vee \mathsf{parent}_u = v$

$\mathsf{C\_Ancestor}(v) \equiv \mathsf{parent}_v \neq \mathsf{suc}_v \wedge \mathsf{parent}_v \neq \mathsf{Pred}(v)$

$\mathsf{Init}(v) \equiv \mathsf{DFS\_F}(v) \wedge \mathsf{DefCycle}[0]_v = \mathsf{label}_v$

$\mathsf{Nds\_Verify}(v) \equiv [(\mathsf{Ask\_V}(v) \wedge \mathsf{VarCycle}_v = (\mathsf{Max\_C}(v), \mathsf{Way\_C}(v))) \vee \mathsf{Ask\_I}(v)] \wedge \mathsf{DefCycle}_v = \mathsf{DefCycle}_{\mathsf{Pred}(v)}$

$\mathsf{NdDel}(v) \equiv \mathsf{state}_{\mathsf{parent}_v} \neq \mathtt{Done} \wedge \mathsf{state}_{\mathsf{parent}_v} \neq \mathtt{Propag} \wedge \neg\mathsf{Improve}(v)$

Figure 4: Corrections predicates used by the algorithm.

**Tree construction** LoopFreeMST starts by constructing a spanning tree of the graph, using the self-stabilizing loop-free algorithm "Tree construction" described in [20]. This algorithm constructs a BFS, and uses two variables *parent* and *distance*. During the execution of our algorithm, these two variables are subject to the same rules as in [20]. After each modification of the spanning tree, the new distance to the parent is propagated in sub-trees by Rules $\mathsf{R_P}$ and $\bar{\mathsf{R}}_{\mathsf{P}}$.

$\mathsf{R_P}$: **(Distance propagation)**
**If** $\mathsf{Coherent\_Done}(v) \wedge \neg\mathsf{Ask\_V}(v) \wedge (\mathsf{state}_{\mathsf{parent}_v} = \mathtt{Improve} \vee \mathsf{state}_{\mathsf{parent}_v} = \mathtt{Propag}) \wedge \mathsf{suc}_v \neq \mathsf{parent}_v \wedge$
$\mathsf{Pred}(v) \neq \mathsf{parent}_v \wedge \mathsf{d}_v \neq \mathsf{d}_{\mathsf{parent}_v} + 1$
**then** $\mathsf{state}_v := \mathtt{Propag}; \mathsf{d}_v := \mathsf{d}_{\mathsf{parent}_v} + 1$;

$\bar{\mathsf{R_P}}$: **(End distance propagation)**
**If** $\mathsf{state}_v = \mathtt{Propag} \wedge \mathsf{EndPropag}(v)$
**then** $\mathsf{state}_v := \mathtt{Done}; \mathsf{DefCycle}_v := (\mathsf{label}_v, \mathsf{done}); \mathsf{VarCycle}_v := (0, \mathsf{Before}); \mathsf{suc}_v := \emptyset;$

**Token circulation and pre-order labeling** LoopFreeMST uses the algorithm described in [4] to provide each node $v$ with a label $\mathsf{label}_v$. Each label is unique in the network traversed by the token. This labeling is used to find the unique path between two nodes in the spanning tree, in a distributed manner. For this purpose, we use the snap-stabilizing algorithm described in [23] for the circulation of a token in the spanning tree. We have slightly modified this algorithm because LoopFreeMST stops the token circulation at a node during the "Cycle improvement" procedure. A node $v$ knows if it has the token by applying predicate $\mathsf{Init}(v)$. Rule $\mathsf{R_{DFS}}$ guides the circulation of the token. The token carries on its tree traversal if one of the following three conditions is satisfied: (i) there is no improvement which could be initiated by the node which holds the token, (ii) an improvement was performed in the current cycle, or (iii) inconsistent node labels were detected in the current cycle. The latter is under the control of Predicate $\mathsf{ContinueDFS}(v)$.

$\mathsf{R_{DFS}}$: **(Continue DFS token circulation)**
**If** $\mathsf{CoherentCycle}(v) \wedge \mathsf{Init}(v) \wedge \mathsf{ContinueDFS}(v)$
**then** $\mathsf{state}_v := \mathtt{Done}; \mathsf{DefCycle}[1]_v = \mathsf{done};$

**Cycle improvement rules** The procedure "Cycle improvement" is the core of LoopFreeMST. Its role is to avoid disconnection of the current spanning tree, while successively improving the tree until reaching a MST. The procedure can be decomposed in four tasks: (1) to check whether the fundamental cycle of the non-tree edge has an improvement or not, (2) perform the improvement if any, (3) update the distances, and (4) resume the token circulation.

Let us start by describing the first task. A node $u$ in state `Done` changes its state to `Verify` if its variables are in consistent state, it has a token, and it has identified a candidate (i.e., an incident non-tree edge $e = \{u, v\}$ whose other extremity $v$ has a smaller label than the one of $u$). The latter is under the control of Predicate $\mathsf{InitVerify}(v)$, and the variable $\mathsf{VarCycle}_v$ contains the label of $u$ and $v$. If the three conditions are satisfied, then the verification of the fundamental cycle $C_e$ is initiated from node $u$, by applying rule $\mathsf{R_V}$. The goal of this verification is twofold: first, to verify whether $C_e$ exists or not, and, second, to save information about the maximum edge weight and the location of the edge of maximum weight in $C_e$. These information are stored in the variable $\mathsf{Way\_C}(v)$. In order to respect the orientation in the current spanning tree, the node $u$ or $v$ that initiates the improvement depends on the localization of the maximum weight edge $f$ in $C_e$. More precisely, let $r$ be the least common ancestor of nodes $u$ and $v$ in the current tree. If $f$ occurs before $r$ in $T$ in the traversal of $C_e$ from $u$ starting by edge $(u, v)$, then the improvement starts from $u$, otherwise the improvement starts from $v$. To get the flavor of our method, let us consider the example depicted on Figure 2. In this example, $f$ occurs after the least common ancestor (node 6). Therefore node 10 atomically swaps its parent to respect the orientation. However, if one replaces in the same example the weight of edge $\{11, 6\}$ by 11 instead of 3, then $f$ would occur before $r$, and thus node 12 would have to atomically swaps its parent. The relative places of $f$ and $r$ in the cycle is indicated by Predicate $\mathsf{Way\_C}(v)$ that returns two different values: Before or After. During the improvement of the tree, the fundamental cycle is modified. It is crucial to save information about this cycle during this modification. In particular, the successor of a node $w$ in a cycle, stored in the variable $\mathsf{suc}_w$, must be preserved. Its value is computed by Predicate $\mathsf{Succ}(v)$ which uses node labels to identify the current examined fundamental cycle. Each node is able to compute its predecessor in the fundamental cycle by applying Predicate $\mathsf{Pred}(v)$. The state of a node is compared with the ones of its successor and predecessor to detect potential inconsistent values. At the end of this task, the node $u$ learns the maximum weight of the cycle $C_e$ and can decide whether it is possible to make an improvement or not. If not, but there is another non-tree edge $e'$ that is candidate for potential replacement, then $u$ verifies $C_{e'}$. Otherwise the token carries on its traversal, and rule $\bar{\mathsf{R_P}}$ is applied.

$\mathsf{R_V}$**:** (Verify rule)

**If** $\mathsf{CoherentCycle}(v) \wedge \neg\mathsf{Error}(v) \wedge (\mathsf{InitVerify}(v) \vee [\neg\mathsf{Init}(v) \wedge (\mathsf{Coherent\_Done}(v) \vee \mathsf{state}_v = \mathtt{Propag}) \wedge \mathsf{Ask\_V}(v)])$
**then** $\mathsf{state}_v := \mathtt{Verify}$;
  **If** $\mathsf{DFS\_F}(v)$ **then** $\mathsf{DefCycle}[1]_v := \mathsf{LabCand}(v)$;
  **Else** $\mathsf{DefCycle}_v := \mathsf{DefCycle}_{\mathsf{Pred}(v)}; \mathsf{VarCycle}_v := (\mathsf{Max\_C}(v), \mathsf{Way\_C}(v)); \mathsf{suc}_v := \mathsf{Succ}(v)$;

$\mathsf{Pred}(v) \equiv \arg\min\{\mathsf{label}_u : u \in N(v) \wedge \mathsf{state}_u \neq \mathtt{Done} \wedge \mathsf{state}_u \neq \mathtt{Propag} \wedge \mathsf{suc}_u = v\}$ if $u$ exists, $\emptyset$ otherwise

$\mathsf{MaxLab}(v,x) \equiv \arg\max\{\mathsf{label}_s : s \in N(v) \wedge \mathsf{label}_s < x\}$

$$\mathsf{Succ}(v) \equiv \begin{cases} \mathsf{VarCycle}[0]_v & \text{if } \mathsf{DefCycle}[1]_v = \mathsf{label}_v \\ \mathsf{parent}_v & \text{if } (\mathsf{label}_v > \mathsf{DefCycle}[1]_v \wedge \mathsf{state}_v = \mathtt{Verify}) \vee (\mathsf{label}_v < \mathsf{DefCycle}[1]_v \wedge \\ & (\mathsf{state}_v = \mathtt{Improve} \vee \mathsf{state}_v = \mathtt{End})) \\ \mathsf{MaxLab}(v, \mathsf{DefCycle}[1]_v) & \text{if } (\mathsf{label}_v < \mathsf{DefCycle}[1]_v \wedge \mathsf{state}_v = \mathtt{Verify}) \\ \mathsf{MaxLab}(v, \mathsf{label}_v) & \text{if } (\mathsf{label}_v > \mathsf{DefCycle}[1]_v \wedge (\mathsf{state}_v = \mathtt{Improve} \vee \mathsf{state}_v = \mathtt{End})) \end{cases}$$

$\mathsf{Max\_C}(v) \equiv \max\{\mathsf{VarCycle}[0]_{\mathsf{Pred}(v)}, w(v, \mathsf{Pred}(v))\}$

$$\mathsf{Way\_C}(v) \equiv \begin{cases} \mathsf{After} & \text{if } \mathsf{VarCycle}[0]_v \neq \mathsf{VarCycle}[0]_{\mathsf{Pred}(v)} \wedge \mathsf{label}_v > \mathsf{label}_{\mathsf{Pred}(v)} \\ \mathsf{VarCycle}[1]_{\mathsf{Pred}(v)} & \text{otherwise} \end{cases}$$

$\mathsf{LabCand}(v) \equiv \min\{\mathsf{label}_u : u \in N(v) \wedge \mathsf{label}_u < \mathsf{label}_v \wedge \neg\mathsf{Tree\_Edge}(v,u) \wedge \mathsf{label}_u \succ \mathsf{DefCycle}[1]_v\}^a$ if $u$ exists, $\mathsf{end}$ otherwise

[a] $\succ$ order on neighbor labels for which 'end' is the biggest element and 'done' is the smallest one.

Figure 5: Predicates used by the algorithm.

If $C_e$ can yield an improvement, then rule $\mathsf{R_I}$ is executed. By this rule, a node enters in state `Improve`, and changes its parent to its predecessor if $\mathsf{VarCycle}[1]_v = \mathsf{Before}$ (respectively to its successor if $\mathsf{VarCycle}[1]_v = \mathsf{After}$). For this purpose, it uses the variable $\mathsf{suc}_v$ and the predicate $\mathsf{Pred}(v)$ .

$\mathsf{R_I}$**: (Improve rule)**

**If** $\mathsf{CoherentCycle}(v) \wedge \neg\mathsf{Error}(v) \wedge \mathsf{Coherent\_Verify}(v) \wedge \mathsf{Improve}(v) \wedge \neg\mathsf{C\_Ancestor}(v) \wedge [(\mathsf{DFS\_F}(v) \wedge \mathsf{Ask\_V}(v)) \vee \mathsf{Ask\_I}(v)]$
**then** $\mathsf{state}_v := \mathtt{Improve}$;
  **If** $\mathsf{DFS\_F}(v) \vee \mathsf{state}_{\mathsf{Pred}(v)} = \mathtt{Improve}$ **then** $\mathsf{VarCycle}_v := \mathsf{VarCycle}_{\mathsf{Pred}(v)}$
  **If** $(\mathsf{DFS\_F}(v) \wedge \mathsf{VarCycle}[1]_v = \mathsf{Before}) \vee \neg\mathsf{DFS\_F}(v)$ **then** $\mathsf{parent}_v := \mathsf{Pred}(v)$;
  **If** $\mathsf{state}_{\mathsf{suc}_v} = \mathtt{Improve}$ **then** $\mathsf{VarCycle}_v := \mathsf{VarCycle}_{\mathsf{suc}_v}; \mathsf{parent}_v := \mathsf{suc}_v$;
  **If** $w(v, \mathsf{suc}_v) \geq \mathsf{VarCycle}[0]_v$ **then** $\mathsf{suc}_v = \mathsf{Succ}(v)$
  $\mathsf{d}_v := \mathsf{d}_{\mathsf{parent}_v} + 1$;

At the end of an improvement, it is necessary to inform the node holding the token that it has to carry on its traversal. This is the role of rule $\mathsf{R_E}$. It is also necessary to inform all nodes impacted by the modification that they have to update their distances to the root (see Section 3.2).

$\mathsf{R_E}$**: (End of improvement rule)**

**If** $\mathsf{CoherentCycle}(v) \wedge \neg\mathsf{Error}(v) \wedge \mathsf{End\_Improve}(v) \wedge \mathsf{EndPropag}(v)$
**then** $\mathsf{state}_v := \mathtt{End}$;

$\mathsf{Candidate}(v) \equiv \mathsf{LabCand}(v) \neq \mathsf{end}$

$\mathsf{InitVerify}(v) \equiv \mathsf{Init}(v) \wedge \mathsf{Candidate}(v) \wedge (\mathsf{Coherent\_Done}(v) \vee [\mathsf{Coherent\_Verify}(v) \wedge \neg\mathsf{Improve}(v) \wedge \neg\mathsf{C\_Ancestor}(v) \wedge \mathsf{Ask\_V}(v)])$

$\mathsf{ImproveF}(v, x) \equiv \neg\mathsf{Tree\_Edge}(v, x)) \wedge \max(\mathsf{VarCycle}[0]_v, \mathsf{VarCycle}[0]_x) > w(v, x)$

$\mathsf{Improve}(v) \equiv \mathsf{ImproveF}(v, \mathsf{Pred}(v)) \vee \mathsf{ImproveF}(v, \mathsf{suc}_v)$

$\mathsf{End\_Improve}(v) \equiv \mathsf{Coherent\_Improve}(v) \wedge (\mathsf{NdDel}(v) \vee \mathsf{Ask\_EI}(v))$

$\mathsf{ContinueDFS}(v) \equiv (\mathsf{Init}(v) \wedge [([\mathsf{Coherent\_Done}(v) \vee (\mathsf{Coherent\_Verify}(v) \wedge \neg\mathsf{ImproveF}(v, \mathsf{Pred}(v)) \wedge \mathsf{Ask\_V}(v))] \wedge \neg\mathsf{Candidate}(v)) \vee \mathsf{Coherent\_End}(v) \vee \mathsf{Error}(v)]) \vee \neg\mathsf{DFS\_F}(v)$

$\mathsf{Error}(v) \equiv \mathsf{state}_v \neq \mathtt{Done} \wedge \mathsf{state}_v \neq \mathtt{Err} \wedge (\mathsf{suc}_v = \mathsf{Succ}(v) = \emptyset \vee \mathsf{Ask\_E}(v))$

$\mathsf{EndPropag}(v) \equiv (\forall u \in N(v), \mathsf{parent}_u = v \wedge \mathsf{state}_u = \mathtt{Done} \wedge \mathsf{d}_u = \mathsf{d}_v + 1)$

Figure 6: Predicates used by the algorithm.

**Module composition** All the different modules presented, except the tree construction parts of the correction module, need the presence of a spanning tree in $G$. Thus, we must execute the tree construction rules first if an incoherency in the spanning tree is detected. To this end, these rules are composed using the level composition defined in [15]. If Predicate $\mathsf{CoherentTree}(v)$ is not verified then the tree construction rules are executed, otherwise the other modules can be executed. The token circulation algorithm and the naming algorithm are composed together using the conditional composition described in [4]. Finally, we compose the token circulation algorithm and the cycle improvement module with a conditional composition using Predicate $\mathsf{ContinueDFS}(v)$ defined in the algorithm. This allows to execute the token circulation algorithm only if the cycle improvement module does not need the token on a node. Figure 7 shows how the different modules are composed together.


Figure 7: Composition of the presented modules.

# 4 Concluding remarks

We presented a new solution to the distributed MST construction that is both self-stabilizing and loop-free. It improves on memory usage from $O(n \log n)$ to $O(\log n)$, yet doesn't make strong system assumptions such as knowledge of network size or unicity of edge weights, making it particularly suited to dynamic networks. Two important open questions are raised:

1. For depth first search tree construction, self-stabilizing solutions that use only constant memory space do exist. It is unclear how the obvious constant space lower bound can be raised with respect to metrics that minimize a global criterium (such as MST).

2. Our protocol pionneers the design of self-stabilizing loop-free protocols for *non* locally optimizable tree metrics. We expect the techniques used in this paper to be useful to add loop-free property for other metrics that are only globally optimizable, yet designing a generic such approach is a difficult task.

## References


[1] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In André Schiper, editor, *WDAG*, volume 725 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 1993.

[2] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing byzantine tolerant digital clock synchronization. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *PODC*, pages 385–394. ACM, 2008.

[3] Jorge Arturo Cobb and Mohamed G. Gouda. Stabilization of general loop-free routing. *J. Parallel Distrib. Comput.*, 62(5):922–944, 2002.

[4] Ajoy Kumar Datta, Shivashankar Gurumurthy, Franck Petit, and Vincent Villain. Self-stabilizing network orientation algorithms in arbitrary rooted networks. *Stud. Inform. Univ.*, 1(1):1–22, 2001.

[5] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[6] S. Dolev. *Self-stabilization.* MIT Press, March 2000.

[7] Shlomi Dolev. *Self-Stabilization.* MIT Press, 2000.

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

[9] Shlomi Dolev and Jennifer L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997.

[10] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, 2004.

[11] Eli M. Gafni and P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 29:11–18, 1981.

[12] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[13] J. J. Garcia-Luna-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Trans. Netw.*, 1(1):130–141, 1993.

[14] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance (preliminary version). In *PODC*, pages 195–206, 1993.

[15] Mohamed G. Gouda and Ted Herman. Adaptive programming. *IEEE Trans. Software Eng.*, 17(9):911–921, 1991.

[16] Mohamed G. Gouda and Marco Schneider. Stabilization of maximal metric trees. In Anish Arora, editor, *WSS*, pages 10–17. IEEE Computer Society, 1999.

[17] Sandeep K. S. Gupta and Pradip K. Srimani. Self-stabilizing multicast protocols for ad hoc networks. *J. Parallel Distrib. Comput.*, 63(1):87–96, 2003.

[18] Lisa Higham and Zhiying Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *DISC*, pages 194–208, 2001.

[19] Colette Johnen and Sébastien Tixeuil. Route preserving stabilization. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2003.

[20] Colette Johnen and Sébastien Tixeuil. Route preserving stabilization. In *Self-Stabilizing Systems*, pages 184–198, 2003.

[21] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[22] Marina Papatriantafilou and Philippas Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997.

[23] Franck Petit and Vincent Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. *J. Parallel Distrib. Comput.*, 67(1):1–12, 2007.

[24] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, pages 1389–1401, 1957.

[25] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

# Appendix

## Correctness proof

We use the algorithm given in [20] to construct a breadth first search spanning tree. Note that, the algorithm given in [20] satisfies the loop-free property. Therefore, in the remainder we suppose there is a constructed spanning tree.

**Theorem 1** (LoopFreeMST) *Starting from an arbitrary spanning tree of the network $G$,* LoopFreeMST *algorithm is a self-stabilizing loop-free algorithm.*

**Proof.** Let $T$ a spanning tree of network $G$ and $v$ a node of $T$. If $v$ is in an incoherent state then according to Lemma 1 below, the algorithm bootstraps the state of $v$, otherwise the token continues its circulation in the tree until a verification on a node is needed (Lemma 3). When the token is on a node that has candidate edges not in the tree (i.e. whose fundamental cycle is not yet checked), according to Corollary 1 the algorithm verifies if an amelioration (see Section 3.1 for the definition of an amelioration) must be performed using these not tree edges and according to Lemma 5 an improvement is performed if an improvement is possible. Moreover, the algorithm performs all possible improvements (Lemma 7) until no improvement is feasable (Lemma 9 and Corollary 2), i.e. a minimum spanning tree is reached.

Starting from a spanning tree $T$ of the network, during the execution of the algorithm no cycle is created and a spanning tree structure is preserved (see Corollary 3). Moreover, according to Lemma 11 if $T$ is minimum spanning tree then $T$ is maintained by the algorithm. □

**Lemma 1 (Bootstrap)** *A node $v$ in an incoherent state for the cycle improvement module eventually verifies the predicate* CoherentCycle($v$)*.*

**Proof.** A node may have six different states in the algorithm: `Done`, `Verify`, `Improve`, end, `Err`, and `Propag`. The coherence of a node in these different states is defined respectively by predicates Coherent_Done, Coherent_Verify, Coherent_Improve, Coherent_End, and Coherent_Error. For the state `Propag`, we detect if the propagation is done using Predicate EndPropag($v$) to allow the execution of Rule $\mathsf{R_D}$ to reinitialize the state of the node. According to the algorithm description, if a node $v$ is not coherent (i.e. does not respect one of the previous mentioned predicates), Predicate CoherentCycle($v$) is not verified since the previous mentioned predicates are exclusive because a node can have one state. Thus, $v$ can execute Rule $\mathsf{R_D}$ to correct its variables to a coherent state satisfying Predicate Coherent_Done($v$). As a consequence Predicate CoherentCycle($v$) is satisfied too (see Rule $\mathsf{R_D}$). □

**Lemma 2** *If* CoherentCycle($v$) $= true$*,* Succ($v$) $= \emptyset$ *and* EndPropag($v$) $= true$ *then eventually a node $v$ is in status* `Err` *and satisfies* Coherent_Error($v$)*.*

**Proof.** We show that if a node $v$ in a fundamental cycle has no successor because of bad labels then $v$ changes its status to `Err`. Predicate Succ($v$) is in charge to give the successor of a node in a fundamental cycle based on the node labels, following Observation 1 below. Thus, if Predicate Succ($v$) returns no successor this implies that bad labels disturb the computation of the successor. Predicate Error($v$) is in charge to detect bad labels. We show that a node $v$ which is part of a fundamental cycle (i.e. satisfies Predicate CoherentCycle($v$)) and detects an error or has its successor in status `Err` changes its status to `Err` (except the initiator node, i.e. $\mathsf{DefCycle}[0]_v \neq \mathsf{label}_v$). We do not consider the status `Done` since in this status no node has a successor (see Predicate Error($v$).

Consider any node $v$ (except the initiator node) which satisfies Predicate $\mathsf{CoherentCycle}(v)$. To change its status to `Err` a node must execute Rule $\mathsf{R_{Err}}$ and we must consider two cases: a node with no successor, or a node with a successor in status `Err`. In the first case, a node $v$ satisfies Predicate $\mathsf{Error}(v)$ (see Predicate $\mathsf{Error}(v)$) and $v$ can execute Rule $\mathsf{R_{Err}}$. After the execution of Rule $\mathsf{R_{Err}}$, $v$ satisfies Predicate $\mathsf{Coherent\_Error}(v)$ since $\mathsf{state}_v = \mathtt{Err}$, $\mathsf{Succ}(v) = \emptyset$ and $\mathsf{DefCycle}_v = \mathsf{DefCycle}_{\mathsf{Pred}(v)}$. In the second case, suppose that for a node $v$ we have $\mathsf{state}_{\mathsf{suc}_v} = \mathtt{Err}$. According to Predicate $\mathsf{Ask\_E}(v)$, $\mathsf{Error}(v) = true$ and thus $v$ can execute Rule $\mathsf{R_{Err}}$ to change its status to `Err`. After the execution of Rule $\mathsf{R_{Err}}$, $v$ satisfies Predicate $\mathsf{Coherent\_Error}(v)$ since $\mathsf{state}_v = \mathtt{Err}$, $\mathsf{Ask\_E}(v) = true$ and $\mathsf{DefCycle}_v = \mathsf{DefCycle}_{\mathsf{Pred}(v)}$. One can show by induction following the same argument that any node part of a fundamental cycle with bad labels changes its status to `Err` (except the initiator node). □

**Lemma 3 (Token circulation)** *Starting from a configuration where a spanning tree $T$ is constructed, if a node $v$ has the DFS token and satisfies* $\mathsf{CoherentCycle}(v)$ *then eventually Predicate* $\mathsf{ContinueDFS}(v)$ *returns true.*

**Proof.** Predicate $\mathsf{ContinueDFS}(v)$ notices when the DFS token must continue its circulation in the tree. The DFS token must continue its circulation in four cases: (1) a node in status `Done` has no candidate edge, (2) a node in status `Verify` with no possible improvement has no candidate edge, (3) an improvement was done in the fundamental cycle, or (4) bad labels are detected in the fundamental cycle.

In case 1, for node $v$, $\mathsf{Coherent\_Done}(v) = true$ (otherwise according to Lemma 1 its state is reinitialized). In case 2, for node $v$, $\mathsf{Coherent\_Verify}(v) = true$ (otherwise according to Lemma 1 its state is reinitialized) and Predicate $\mathsf{ImproveF}(v)$ is used to detect possible improvements (see the proof of Lemma 4). For case 1 and 2, if $v$ has no candidate Predicate $\mathsf{Candidate}(v) = false$ (see Predicate $\mathsf{Candidate}(v)$ and proof of Lemma 4) and thus Predicate $\mathsf{ContinueDFS}(v)$ is satisfied. In case 3, according to Lemma 6 the initiator node $v$ satisfies Predicate $\mathsf{Coherent\_End}(v)$ and Predicate $\mathsf{ContinueDFS}(v)$ returns true. Finally in case 4, according to Lemma 2 the successor of the initiator node $v$ is in status `Err` so Predicate $\mathsf{Ask\_E}(v) = true$ and Predicate $\mathsf{Error}(v)$ returns true. Thus, Predicate $\mathsf{ContinueDFS}(v)$ returns true.

Therefore, in all the above cases Predicate $\mathsf{ContinueDFS}(v)$ returns true and $v$ can execute Rule $\mathsf{R_{DFS}}$ to allow the token circulation. It then changes its status to Done and sets $\mathsf{DefCycle}[1]_v$ to done to force the verification of all adjacent candidate edges in the next tree traversal by the DFS token. □

**Observation 1** *Let $T$ be a tree spanning $V$ and correctly labeled. Let an edge $e = \{u, v\} \in E, e \notin T$, $C_e$ its fundamental cycle and $x$ the fundamental cycle root of $C_e$. There is always a path $P(u, v)$ in $T$ between $u$ and $v$, such that $P(u, v)$ can be decomposed in two parts: a sub-path $P(x, u) \subset P(u, v)$ (resp. $P(x, v) \subset P(u, v)$) with increasing labels from $x$ to $u$ (resp. $x$ to $v$).*

**Lemma 4 (Cycle verification)** *Let $v$ be a node of $T$ such that $v$ has the DFS token with at least an adjacent edge $e = \{u, v\} \in E, e \notin T$ whose fundamental cycle is not already verified by the algorithm. Eventually the cycle improvement module verifies if there is an improvement in $C_e$.*

**Proof.** Suppose first that $v$ has the DFS token and $v$ is in a coherent state `Done`, otherwise according to Lemma 1 its state is corrected. Let $e = \{u, v\}$ be a not tree edge, which is a candidate edge for $v$, i.e., we have $\mathsf{Candidate}(v) \neq end$. We consider that $\mathsf{label}_u < \mathsf{label}_v$ since a candidate edge for node $v$ is an adjacent not tree edge $e = \{u, v\}$ with $\mathsf{label}_u < \mathsf{label}_v$, see predicate $\mathsf{LabCand}(v)$. Since $v$ is in a coherent state `Done` and $\mathsf{Candidate}(v) \neq end$, we have variable $\mathsf{DefCycle}[1]_v$ equal to $\mathsf{done}$, Predicate $\mathsf{CoherentCycle}(v)$ and $\mathsf{InitVerify}(v)$ return true, whears Predicate $\mathsf{Error}(v)$ returns false. Thus, $v$ can execute Rule $\mathsf{R_V}$. Note that Rule $\mathsf{R_{DFS}}$ can not be executed since Predicate $\mathsf{ContinueDFS}(v)$ returns false since $\mathsf{Candidate}(v) \neq end$. As a consequence $v$ stops the DFS token and becomes the initiator node of cycle $C_e$ with $u$ as target node (see Rule $\mathsf{R_V}$).

After the execution of Rule $\mathsf{R_V}$, $v$ is in state `Verify` and according to predicate $\mathsf{Succ}(v)$ $v$ selects its father as next node in the cycle (i.e. $\mathsf{suc}_v = \mathsf{parent}_v$). Note that since $v$ is in coherent state `Done` variable $\mathsf{VarCycle}_v = (0, \mathsf{Before})$. Cycle $C_e$ is decomposed in two parts (see Lemma 1): (1) from the initiator $v$ to the root $x$ of $C_e$ and (2) from $x$ to the target node $u$. In the following we prove by induction on the length of cycle $C_e$ that a node $a$ belonging to $C_e$ executes Rule $\mathsf{R_V}$ and eventually is in state `Verify`. Moreover, variable $\mathsf{suc}_a$ describes the successor of $a$ in $C_e$ (i.e. encodes the cycle $C_e$).

Case 1: Consider a coherent node $a$ in state `Done` (see Lemma 1) which has not the DFS token (i.e. Predicate $\mathsf{Init}(a)$ is false). Consider the successor node of $C_e$'s initiator node $v$. As described above, $v$ is in state `Verify` and $\mathsf{suc}_v = a$. According to Predicate $\mathsf{Pred}(a)$, $v$ is the predecessor of $a$ in cycle $C_e$ since $a$ is the parent of $v$ in the tree. Thus, Predicate $\mathsf{Ask\_V}(a)$ returns true and $a$ could execute Rule $\mathsf{R_V}$. Therefore, $a$ is in state `Verify` and selects its parent as its successor in $C_e$, like $v$. Moreover, $a$ computes the new heaviest edge from $v$ to $a$ and notices that the heaviest edge location is before (i.e. $\mathsf{Before}$) the root of $C_e$ (see respectively predicates $\mathsf{Max\_C}$ and $\mathsf{Way\_C}$). Using the same scheme, we can show that all nodes on $C_e$ between $v$ and $x$ (including $x$) execute Rule $\mathsf{R_V}$ and are in state `Verify`.

Case 2: Consider a coherent node $a$ in state `Done` (see Lemma 1) which has not the DFS token (i.e. Predicate $\mathsf{Init}(a)$ is false) and is the successor node of $x$. As described in case 1, $x$ is in state `Verify`. Since $x$ is the parent of $a$ in the tree, Predicate $\mathsf{Pred}(a)$ returns $x$ as predecessor of $a$. Thus, Predicate $\mathsf{Ask\_V}(a)$ returns true and $a$ can execute Rule $\mathsf{R_V}$. $a$ selects as its successor in $C_e$ the child with the highest label smaller than target node's $u$ label (see predicates $\mathsf{MaxLab}(a)$ and $\mathsf{Succ}(a)$). Moreover, $a$ computes the new heaviest edge from $v$ to $a$ and if $a$ has a different heaviest edge $a$ notice that the heaviest edge location is after (i.e. $\mathsf{After}$) the root of $C_e$ otherwise $a$ takes the location of its predecessor (see respectively predicates $\mathsf{Max\_C}$ and $\mathsf{Way\_C}$). Using the same scheme, we can show that all nodes on $C_e$ between $x$ and $u$ (including $u$) execute Rule $\mathsf{R_V}$ and are in state `Verify`. Note that the target node $u$ selects $v$ as its successor in $C_e$ (see Predicate $\mathsf{Succ}(u)$).

Consider now that $v$ has the DFS token, is in a coherent state `Verify` and predecessor of $v$ is in state `Verify` (i.e. $\mathsf{Ask\_V}(v) = true$). Note that the predecessor of $v$ is the target node $u$. As described in case 2, target node $u$ knows the weight of the heaviest edge $e'$ in $C_e$ ($e' \in T$). Thus, $v$ could check if there is an improvement in $C_e$ (see Predicate $\mathsf{Improve}(v)$). □

**Corollary 1 (Node cycles verification)** *Let $T$ a spanning tree and $v$ be a node of $T$ such that $v$ has the DFS token. Eventually for each adjacent candidate edge $e$ of $v$, the cycle improvement module verifies if there is an improvement in $C_e$.*

**Proof.** We prove that while there is no improvement initiated by $v$, each edge $e = \{u, v\} \in E, e \notin T$ is eventually examined by the cycle improvement module. We consider the two cases below: (1) there is no improvement initiated by $v$, or (2) an improvement can be done in $C_e$ for a candidate edge $e$. Consider an arbitrary candidate edge $e = \{u, v\} \in E, e \notin T$. According to Lemma 4, $v$ eventually verifies if there is an improvement in $C_e$.

Case 1: If there is no improvement in $C_e$ and $v$ has another candidate edge (i.e. predicates $\mathsf{Candidate}(v)$ and $\mathsf{InitVerify}(v)$ return true) then $v$ must check if there is an improvement in the fundamental cycle of the new candidate edge. According to Lemma 4, $v$ could execute again Rule $\mathsf{R_V}$ with a new target and stay in a coherent state `Verify`. Therefore for each not tree adjacent edge $e$, $v$ eventually verifies if there is an improvement in the fundamental cycle $C_e$.

Case 2: If an improvement can be done in $C_e$, when the improvement is done, $v$ is in the state `End`. Thus, Predicate $\mathsf{ContinueDFS}(v)$ returns true and Rule $\mathsf{R_{DFS}}$ can be executed to continue the token circulation in the tree. However, the next time $v$ has the token as described in case 1, $v$ eventually checks again the previously examined edges, but $v$ will also check candidate edges not previously visited.

□

**Definition 4 (Red Rule)** *If $C$ is a cycle in $G = (V, E)$ with no red edges then color in red the maximum weight edge in $C$.*

**Theorem 2 (Tarjan et al. [25])** *Let $G$ be a connected graph. If it is not possible to apply Red Rule then the set of not colored edges forms a minimum spanning tree of $G$.*

**Lemma 5 (Improvement)** *Let an edge $e = \{u, v\} \in E, e \notin T$ and let $C_e$ its fundamental cycle. If there exists a possible improvement in $C_e$ then the algorithm eventually performs the improvement.*

**Proof.** According to Definition 4, there is an improvement in a cycle $C$ if the edge of maximum weight in $C$ belongs to the current tree and one can use the Red Rule. Given an edge $e = \{u, v\} \in E, e \notin T$ and $C_e$ its fundamental cycle, Lemma 4 states that the initiator node $v$ detects if there is an improvement in cycle $C_e$. Assume that an improvement can be performed in cycle $C_e$ (i.e. predicate $\mathsf{Improve}(v) = true$). As proved in Lemma 4, $u$ and $v$ are in a coherent state `Verify` and have a successor, thus we have $\mathsf{CoherentCycle}(v) = true, \mathsf{Error}(v) = false$ and $\mathsf{Ask\_V}(v) = true$. Since $v$ is the initiator node of $C_e$, $v$ has the DFS token and could not be the root of $C_e$ (i.e. $\mathsf{DFS\_F}(v) = true$ and $\mathsf{C\_Ancestor}(v) = false$). So $v$ can execute Rule $\mathsf{R_I}$, to change its state to `Improve` and to update its estimation of the heaviest edge of $C_e$ and the heaviest edge location to the values of its predecessor (i.e. the target node $u$). Two cases have to be analyzed: (1) the heaviest edge location is between $v$ and $x$ (i.e. $\mathsf{VarCycle}[1]_v = \mathsf{Before}$) or (2) between $u$ and $x$ (i.e. $\mathsf{VarCycle}[1]_v = \mathsf{After}$). In the two cases, the improvement must be propagated from $v$ to $x$ (resp. $u$ to $x$) until reaching the (first) heaviest edge or the root of $C_e$ (if the weight of the heaviest edge has been reduced). Indeed, the root of $C_e$ must not change its parent to a neighbor in $C_e$ otherwise it disconnects its subtree from the rest of the tree.

Case 1: Since $\mathsf{VarCycle}[1]_v = \mathsf{Before}$, $v$ takes as new parent its predecessor in the cycle. Let $a$ be a node in coherent state `Verify` between $v$ and $x$ (Note: $a$ exists otherwise suppose $a$ is in an incoherent state, according to Lemma 1 $a$ reinitiates its state to `Done` which induces a propagation of state `Done` in $C_e$, since the nodes are no more coherent with their predecessors, and stops the improvement until a new verification of $C_e$ is restarted). If the improvement must continue (i.e. Predicate $\mathsf{Improve}(a)$ returns true), $a$ is not the root of $C_e$ and its predecessor is in state `Improve` (see Predicate $\mathsf{Ask\_I}$) then $a$ can execute Rule $\mathsf{R_I}$. So, $a$ changes its state to `Improve`, updates its variable $\mathsf{VarCycle}_a$ to the value of its predecessor and takes its predecessor as its parent. This propagation continues until reaching a node $a$ which stops the improvement (i.e. $\mathsf{Improve}(a) = false$ or $\mathsf{C\_Ancestor}(a) = true$).

Case 2: $\mathsf{VarCycle}[1]_v = \mathsf{After}$ and as in case 1 $v$ executes Rule $\mathsf{R_I}$ but $v$ changes only its state to `Improve` and updates its variable $\mathsf{VarCycle}_v$ to the value of its predecessor. Hence $v$ does not change its parent. Consider the target node $u$, we have $\mathsf{Ask\_I}(u) = true$ since $v$ is in state `Improve`. So, $u$ executes Rule $\mathsf{R_I}$, changes its state to `Improve`, updates $\mathsf{VarCycle}_u$ to its successor value and changes its parent to its successor (i.e. $\mathsf{parent}_u = v$). As described in case 1, the improvement is propagated in the cycle from $u$ to $x$ until a node $a$ is reached which stops the improvement (i.e. $\mathsf{Improve}(a) = false$ or $\mathsf{C\_Ancestor}(a) = true$).

Overall, if an improvement exists then this improvement is eventually performed. □

**Lemma 6** *If $v$ satisfies* $\mathsf{Coherent\_Improve}(v)$ *and* $\mathsf{EndPropag}(v)$ *then $v$ eventually changes its status to* `End` *and the predicate* $\mathsf{Coherent\_End}(v)$ *is satisfied.*

**Proof.** We conduct the proof by induction on the length of the fundamental cycle. A node involved in an improvement executes Rule $\mathsf{R_E}$ to inform its predecessor or successor the end of the improvement. An improvement can be propagated by a successor or a predecessor in the cycle. We show the lemma considering that the improvement is propagated by the successor of a node, but the same idea can be applied by considering predecessor instead of successor. Moreover, we assume that labels are correct in the fundamental cycle otherwise it is not necessary to inform the end of the improvement since according to Lemma 2 the nodes are in state `Err`. Let $x$ the node which detects the end of the improvement and $y$ the initiator node in a fundamental cycle.

Consider the node $x$, such that $\mathsf{Coherent\_Improve}(x) = true$ and $w(x, \mathsf{suc}_x) \geq \mathsf{VarCycle}[0]_x$. Predicate $\mathsf{End\_Improve}(x) = true$ since $\mathsf{Coherent\_Improve}(x) = true$ and $\mathsf{NdDel}(x)$ is satisfied because $\mathsf{Improve}(x) = false$. Thus, $x$ can execute Rule $\mathsf{R_E}$ and changes its status to `End`. Therefore, $\mathsf{Coherent\_End}(x)$ is satisfied since $\mathsf{state}_x = \texttt{End}$, $\mathsf{NdDel}(x) = true$ and $\mathsf{DefCycle}_x = \mathsf{DefCycle}_{\mathsf{parent}_x}$ because $x$ and its parent are in the same fundamental cycle. Now, suppose by induction hypothesis that any node $u$ between $x$ and the initiator node $y$ are in state `End` and $\mathsf{Coherent\_End}(u)$ is satisfied. Consider the initiator node $y$, $\mathsf{state}_y = \texttt{Improve}$, $\mathsf{Coherent\_Improve}(y) = true$ and $\mathsf{state}_{\mathsf{suc}_y} = \texttt{End}$. Predicate $\mathsf{End\_Improve}(y)$ is satisfied because Predicate $\mathsf{Ask\_EI}(y) = true$ since $\mathsf{state}_{\mathsf{suc}_y} = \texttt{End}$ and $\mathsf{DefCycle}_y = \mathsf{DefCycle}_{\mathsf{suc}_y}$. Thus, $y$ can execute Rule $\mathsf{R_E}$ and changes its status to `End`. Therefore, Predicate $\mathsf{Coherent\_End}(y)$ is satisfied since $\mathsf{state}_y = \texttt{End}$, $\mathsf{Ask\_EI}(y) = true$ and $\mathsf{DefCycle}_y = \mathsf{DefCycle}_{\mathsf{parent}_y}$ because $y$ and its parent are in the same fundamental cycle. □

**Lemma 7 (MST construction)** *Given a spanning tree $T$, the cycle improvement module performs an improvement if $T$ is not a minimum spanning tree of $G$.*

**Proof.** According to the token circulation algorithm [23], eventually each node in the tree is visited and holds the token. Consider a node $v$ in the tree $T$, which has the DFS token. According to Corollary 1 eventually each adjacent candidate edge of $v$ is examined by the cycle improvement module. Thus, if an improvement is possible this one is detected according to Lemma 4 and performed by $v$ according to Lemma 5. Therefore, if an improvement is possible the cycle improvement module performs it. □

**Lemma 8** *Let $T$ be an existing minimum spanning tree of $G$. The algorithm performs no improvement.*

**Proof.** Let $T$ be an existing minimum spanning tree of $G$ and $v$ be a node in $T$ which has the DFS token. Let $e = \{u, v\}, e \notin T$ an adjacent candidate edge of $v$ and $C_e$ its corresponding fundamental cycle. Suppose the cycle improvement module performs an improvement in $C_e$. We prove by contradiction that no improvement could performed by the algorithm.

Let $w(C_e)$ the maximum edge weight in $C_e$, excluding edge $e$. According to Definition 4, to initiate an improvement from $v$ the following condition must be verified: $w(C_e) > w(e)$.

According to Lemma 4, the predecessor $u$ of $v$ holds the maximum edge weight in $C_e$ (i.e. $\mathsf{VarCycle}[0]_u = w(C_e)$). To perform an improvement, Predicate $\mathsf{Improve}(v)$ must return true to allow $v$ to execute Rule $\mathsf{R_I}$. This implies that $\max(\mathsf{VarCycle}[0]_v, \mathsf{VarCycle}[0]_u) > w(u, v)$ (see Predicate $\mathsf{Improve}(v)$), i.e. $w(C_e) > w(u, v)$ (since $\mathsf{VarCycle}[0]_u = w(C_e)$) which contradicts the fact that no improvement can be performed in $C_e$. Therefore, $v$ can not execute Rule $\mathsf{R_I}$ if no improvement is possible in a fundamental cycle. □

**Corollary 2 (MST conservation)** *Let $T$ be an existing minimum spanning tree of $G$. The algorithm maintains a spanning tree.*

**Proof.** Lemma 8 shows that no improvement is performed by the algorithm if $T$ is a minimum spanning tree of $G$, i.e. Rule $\mathsf{R_I}$ can not be executed by a node. Therefore, according to Lemma 8 and by Remark 1 a spanning tree is maintained. □

**Lemma 9 (Convergence)** *Starting from an illegitimate configuration eventually the algorithm reaches in a finite time a legitimate configuration.*

**Proof.** If the initial configuration contains no spanning tree, there is a node $v$ such that Predicate $\mathsf{CoherentTree}(v) = false$ and according to the level composition (defined in [15]) we use the algorithm given in [20] to construct a breadth first search spanning tree. Otherwise, the initial configuration contains a spanning tree which is not a minimum spanning tree. According to Lemma 7 and 8, improvements are performed by the cycle improvement module until a minimum spanning tree is reached. Moreover, according to Lemma 10 a spanning tree is preserved by the cycle improvement module. Finally, there is at most $m-n+1$ fundamental cycles in any graph so a finite number of improvements can be performed by the cycle improvement module. Thus, in a finite time the algorithm returns a minimum spanning tree. □

**Remark 1** *According to the cycle improvement module description, only Rule $\mathsf{R_I}$ could change the parent of a node.*

**Lemma 10** *Let $T$ be an existing tree spanning $V$, no move performed by the cycle improvement module disconnects $T$.*

**Proof.** There is two cases in which the existing tree $T$ spanning $V$ is disconnected. It is necessary (1) to delete an edge of $T$ by changing the parent of a node (except the root of $T$) to itself or (2) to attribute as new parent of a node a neighbor belonging to its descendant in the tree. Consider the execution of Rule $\mathsf{R_I}$ (see Remark 1). Rule $\mathsf{R_I}$ can be executed by a node if this one is in state `Verify` and is a coherent node (see predicate $\mathsf{Coherent\_Verify}$ in Rule $\mathsf{R_I}$). As described in the proof of Lemma 4, a coherent node in state `Verify` has a predecessor and a successor in a fundamental cycle, note that the initiator has a predecessor because it must wait that this one (i.e. the target node) is in state `Verify` to execute Rule $\mathsf{R_I}$ (see predicate $\mathsf{Ask\_V}$).

Case (1) is not permitted by the algorithm. The new parent of a node is its predecessor or successor in the fundamental cycle (see Rule $\mathsf{R_I}$). Thus the algorithm selects as new parent another node different of the node itself.

Case (2) is not permitted by the algorithm, since the new parent of a node executing Rule $\mathsf{R_I}$ is its predecessor or successor in the fundamental cycle and the edge between the node and its new parent is not already in the tree (see predicate $\mathsf{Improve}$). In other words, the algorithm adds and deletes two adjacent edges in the fundamental cycle, which gives after each move a new spanning tree. Moreover, the algorithm can not change the parent of a fundamental cycle root (see predicate C_Ancestor in guard of Rule $R_I$), in particular the root of the tree, otherwise the subtree of the fundamental cycle root could be disconnected from the rest of the tree. Thus, the new parent is an ancestor or another node with the same ancestor in the tree.

Therefore, after each move performed by the algorithm a spanning tree is preserved. □

**Corollary 3 (Loop-free property)** *Let $T$ be an existing tree spanning $V$, after any move performed by the cycle improvement module $Cycle(T, u, v) = false, \forall u, v \in V$.*

**Proof.** In a configuration where a spanning tree $T$ is constructed, we have $Cycle(T, u, v) = false, \forall u, v \in V$ otherwise it contradicts the fact that $T$ is a spanning tree. Moreover, according to Case (2) in the proof of Lemma 10 any move of the cycle improvement module preserves a spanning tree structure. Thus, for any move $Cycle(T, u, v) = false, \forall u, v \in V$. □

**Lemma 11 (Closure)** *Starting from a legitimate configuration the algorithm preserves a legitimate configuration.*

**Proof.** Let $T$ be an existing tree spanning $V$, such that $T$ is a minimum spanning tree of $G$. Thus, $\forall v \in V$, CoherentTree$(v) = true$. According to the level composition (defined in [15]), since on a node $v$ the predicate CoherentTree$(v)$ determines if the tree must be reconstructed, the only modules executed are the token circulation with labeling module given respectively in [23, 4] and the cycle improvement module. The conditional composition (defined in [4]) between the token circulation with labeling module and the cycle improvement module, using Predicate ContinueDFS$(v)$ on a node $v$ determines which module has to be executed. According to Lemma 3, for any node $v \in V$ eventually Predicate ContinueDFS$(v) = true$ and the DFS token continue its circulation. Otherwise, only the cycle improvement module is executed. According to Lemma 8 and Corollary 2, a minimum spanning tree of $G$ is preserved by the cycle improvement module and therefore by the algorithm composed of the different modules. □

## Complexity

**Lemma 12** *Starting from a configuration where an arbitrary spanning tree is constructed, in at most $O(mn)$ rounds the cycle improvement module produces a minimum spanning tree of $G$, with respectively $m$ and $n$ the number of edges and nodes of the network $G$.*

**Proof.** In a given network $G = (V, E)$, if a spanning tree of $G$ is constructed then there are exactly $m - (n - 1)$ fundamental cycles in $G$ since there are $n - 1$ edges in any spanning tree of $G$. Thus, a tree edge can be contained in at most $m - n + 1$ fundamental cycles. Consider a configuration where a spanning tree $T$ of $G$ is constructed and a tree edge $e_0$ is contained in $m - n + 1$ fundamental cycles and all tree edges have a weight equal to 1, except $e_0$ of weight $w(e_0) > 1$. Suppose that $T$ is not a minimum spanning tree of $G$ such that $\forall e_i \in E, i = 1, \dots, m - n + 1, w(e_{i-1}) > w(e_i)$ with $e_0 \in T$ and $\forall i = 1, \dots, m - n + 1, e_i \notin T$ and $w(e_i) > 1$ (see the graph of Figure 8(a)). Consider the following sequence of improvements: $\forall i, i = 1, \dots, m - n + 1$, exchange the tree edge $e_{i-1}$ by the not tree edge $e_i$ (see a sequence of improvements in Figure 8). In this sequence, we have exactly $m - n + 1$ improvements and this is the maximum number of improvements to obtain a minimum spanning tree since there are $m - n + 1$ fundamental cycles and for each one we apply the Red rule (see Definition 4 and Theorem 2). An improvement can be initiated in the cycle improvement module by a node with the DFS token. The DFS token performs a tree traversal in $O(n)$ rounds. Moreover, each

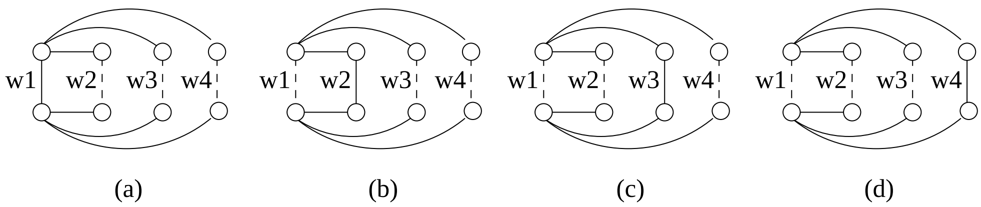


Figure 8: (a) a spanning tree with plain lines in a graph with $m - n + 1$ improvements, (b) the spanning tree obtained after the first improvement, (c) the spanning tree obtained after the second improvement, (d) the minimum spanning tree of the graph obtained after the third improvement.

improvement needs to cross a cycle a constant number of times and each cross requires $O(n)$ rounds. Since at most $m-n+1$ improvements are needed to obtain a minimum spanning tree, at most $O(mn)$ rounds are needed to construct a minimum spanning tree. □

**Lemma 13** *Starting from a legitimate configuration, after a weight edge modification the system reaches a legitimate configuration in at most $O(mn)$ rounds.*

**Proof.** After a weight edge change the system is no more in a legitimate configuration in the following cases: (1) the weight of a not tree edge is less than the weight of the heaviest tree edge in its fundamental cycle, or (2) the weight of a tree edge is greater than the weight of a not tree edge in one of the fundamental cycles including the tree edges.

In each case above, the algorithm must verify if improvements must be performed to reach again a legitimate configuration, otherwise the system is still in a legitimate configuration. Thus, in case (1) it is only sufficient to verify if an improvement must be performed in the fundamental cycle associated to the not tree edge (i.e. to apply the Red rule a single time). To this end, its fundamental cycle must be crossed at most three times: the first time to verify if an improvement is possible, a second time to perform the improvement and a last time to end the improvement, each one needs at most $O(n)$ rounds. According to Lemma 5 and 6, the improvement is performed by the algorithm which leads to a legitimate configuration. Case (2) is more complicated, indeed the weight of a tree edge can change which leads to a configuration where at most $m-n+1$ improvements must be performed to reach a legitimate configuration, since a tree edge can be contained in at most $m-n+1$ fundamental cycles as described in proof of Lemma 12. Since each improvement phase needs $O(n)$ rounds (see case (1)) at most $O(mn)$ rounds are needed to reach a legitimate configuration.

The complexity of case (2) dominates the complexity of the first case. Therefore, after a weight edge change at most $O(mn)$ rounds are needed to reach a legitimate configuration. □