# Fast Self-Stabilizing Minimum Spanning Tree Construction Using Compact Nearest Common Ancestor Labeling Scheme

Lélia Blin, Shlomi Dolev, Maria Gradinariu Potop-Butucaru, Stephane Rovedakis

# Fast Self-Stabilizing Minimum Spanning Tree Construction Using Compact Nearest Common Ancestor Labeling Scheme *

Lélia Blin
Université d'Evry-Val d'Essonne, 91000 Evry, France
LIP6-CNRS UMR 7606, France.
lelia.blin@lip6.fr

Shlomi Dolev
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel.
dolev@cs.bgu.ac.il

Maria Gradinariu Potop-Butucaru
Université Pierre & Marie Curie - Paris 6, 75005 Paris, France.
LIP6-CNRS UMR 7606, France.
maria.gradinariu@lip6.fr

Stéphane Rovedakis
Laboratoire CEDRIC, CNAM, 292 Rue St Martin, 75141 Paris, France.
stephane.rovedakis@cnam.fr

**Abstract**

We present a novel self-stabilizing algorithm for minimum spanning tree (MST) construction. The space complexity of our solution is $O(\log^2 n)$ bits and it converges in $O(n^2)$ rounds. Thus, this algorithm improves the convergence time of previously known self-stabilizing asynchronous MST algorithms by a multiplicative factor $\Theta(n)$, to the price of increasing the best known space complexity by a factor $O(\log n)$. The main ingredient used in our algorithm is the design, for the first time in self-stabilizing settings, of a labeling scheme for computing the nearest common ancestor with only $O(\log^2 n)$ bits.

*A preliminary version of this paper has appeared in the proceedings of the 24th International Conference on Distributed Computing (DISC 2010), see [3].

# 1 Introduction

Since its introduction in a centralized context [22, 17], the minimum spanning tree (or MST) problem gained a benchmark status in distributed computing thanks to the seminal work of Gallager, Humblet and Spira [9].

The emergence of large scale and dynamic systems revives the study of scalable algorithms. A *scalable* algorithm does not rely on any global parameter of the system (e.g. upper bound on the number of nodes or the diameter).

In the context of dynamic systems, after a topology change a minimum spanning tree previously computed is not necessarily a minimum one (e.g., an edge with a weight lower than the existing edges can be added). A mechanism must be used to replace some edges from the constructed tree by edges of lower weight. Park et al. [18, 19] proposed a distributed algorithm to maintain a MST in a dynamic network using the Gallager, Humblet and Spira algorithm. In their approach, each node know its ancestors and the edges weight leading to the root in the tree. Moreover, the common ancestor between two nodes in the tree can be identified. For each non-tree edge $(u, v)$, the tree is detected as not optimal by $u$ and $v$ if there exist a tree edge with a higher weight than $w(u, v)$ between $u$ (resp. $v$) and the common ancestor of $u$ and $v$. In this case, the edge of maximum weight on this path is deleted. This yields to the creation of several sub-trees, from which a new MST can be constructed following the merging procedure given by Gallager et al. [9]. Flocchini et al. [26, 27] considered another point of view to address the same problem. The authors were interested to the problem of precomputing all the replacement minimum spanning trees when a node or an edge of the network fails. They proposed the first distributed algorithms to efficiently solve each of these problems (i.e., by considering either node or edge failure). Additional techniques and algorithms related to the construction of light weight spanning structures are extensively detailed in [21].

Large scale systems are often subject to transient faults. *Self-stabilization* introduced first by Dijkstra in [5] and later publicized by several books [6, 24] deals with the ability of a system to recover from catastrophic situation (i.e., the global state may be arbitrarily far from a legal state) without external (e.g. human) intervention in finite time.

Although there already exist self-stabilizing solutions for the MST construction, none of them considered the extension of the Gallager, Humblet and Spira algorithm (GHS) to self-stabilizing settings. Interestingly, this algorithm unifies the best properties for designing large scale MSTs: it is fast and totally decentralized and it does not rely on any global parameter of the system. Our work proposes an extension of this algorithm to self-stabilizing settings. Our extension uses only poly-logarithmic memory and preserves all the good characteristics of the original solution in terms of convergence time and scalability.

Antonoiu and Srimani, and Gupta and Srimani presented in [10, 11] the first self-stabilizing algorithm for the MST problem. The MST construction is based on the computation of all shortest paths (for a certain cost function) between all 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 $(u, v)$ placed in the MST for each node $v \neq u$. Therefore the algorithm requires $\Omega(\sum_{v \neq u} \log w(u, v))$ 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. The main drawback of this solution is its lack of scalability since each node has to know and maintain information for all the nodes in the system. Note that the authors introduce a time complexity definition related to the transmission of beacon in the context of ad-hoc networks. In a round, each node receives a beacon from all its neighbors. So, the $O(n)$ time complexity announced by the authors stays only in the

particular synchronous settings. In asynchronous setting, a node is activated at the reception of a beacon from each neighbor leading to a $O(n^2)$ time complexity. A different approach for the message-passing model was proposed by Higham and Liang [13]. The algorithm works roughly as follows: every edge checks whether it should belong to the MST or not. To this end, every non tree-edge $e$ floods the network to find a potential cycle, and when $e$ receives its own message back along a cycle, it uses the 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 memory used by each node is $O(\log n)$ bits, but the information exchanged between neighboring nodes is of size $O(n \log n)$ bits, thus only slightly improving that of [10]. This solution also assumes that each node has access to a global parameter of the system: the diameter. Its computation is expensive in large scale systems and becomes even harder in dynamic settings. The time complexity of this approach is $O(mD)$ rounds where $m$ and $D$ are the number of edges and the upper bound of the diameter of the network respectively, i.e., $O(n^3)$ rounds in the worst case.

In [4] we proposed a self-stabilizing loop-free algorithm for the MST problem. Contrary to previous self-stabilizing MST protocols, this algorithm does not make any assumption on the network size (including upper bounds) or the uniqueness of the edge weights. The proposed solution improves on the memory space usage since each participant needs only $O(\log n)$ bits while preserving the same time complexity as the algorithm in [13].

Clearly, in the self-stabilizing implementation of the MST algorithms there is a trade-off between the memory complexity and their time complexity (see Table 1). The challenge we address in this paper is to design fast and scalable self-stabilizing MST with little memory. Our approach brings together two worlds: the time efficient MST constructions and the memory compact informative labeling schemes. We do this by extending the GHS algorithm to the self-stabilizing setting while keeping it memory space compact, but using a self-stabilizing extension of the nearest common ancestor labeling scheme [20, 1]. Note that labeling schemes have already been used in order to maintain compact information linked with vertex adjacency, distance, tree ancestry or tree routing [2], however none of these schemes have been studied in self-stabilizing settings (except for the tree routing).

Our contribution is therefore twofold. We propose for the first time in self-stabilizing settings a $O(\log^2 n)$ bits scheme for computing the nearest common ancestor. Furthermore, based on this scheme, we describe a new self-stabilizing algorithm for the MST problem. Our algorithm does not make any assumption on the network size (including upper bounds) or the existence of an a priori known root. The convergence time is $O(n^2)$ asynchronous rounds and the memory space per node is $O(\log^2 n)$ bits. Interestingly, our work is the first to prove the effectiveness of an informative labeling scheme in self-stabilizing settings and therefore opens a wide research path in this direction. The description of our algorithm is *explicit*, in the sense that we describe all procedures using the formal framework

$$\langle \text{label} \rangle : \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle.$$

The recent paper [16] announces an improvement of our results, by sketching the *implicit* description of a self-stabilizing algorithm for MST converging in $O(n)$ rounds, with a memory of $O(\log n)$ bits per node. This algorithm is also based on an informative labeling scheme. The approach proposed by Korman et al. [16] is based on the composition of many sub-algorithms (some of them not stabilizing) presented in the paper as black boxes and the composition of all these modules was not proved formally correct in self-stabilizing settings up to date. The main feature of our solution in comparison with [16] is its straightforward implementation.

| | a priori knowledge | space complexity | convergence time |
|---|---|---|---|
| [10] | network size and the nodes in the network | $O(n \log n)$ | $O(n^2)$ |
| [13] | upper bound on diameter | $O(\log n)$ messages of size $O(n \log n)$ | $O(n^3)$ |
| [4] | none | $O(\log n)$ | $O(n^3)$ |
| This paper | none | $O(\log^2 n)$ | $O(n^2)$ |

Table 1: Distributed Self-Stabilizing algorithms for the MST problem

# 2 Model and overview of our solution

## 2.1 Model

We consider an undirected weighted connected network $G = \langle V, E, w \rangle$ where $V$ is the set of nodes, $E$ is the set of edges and $w : E \to \mathbb{R}^+$ is a positive cost function. Nodes represent processors and edges represent bidirectional communication links.

The processors asynchronously execute their programs consisting of a set of variables and a finite set of rules. We consider the local shared memory model of computation[1]. 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 *enabled* and may execute 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 this context, a *round* is the smallest portion of an execution where every process has the opportunity to execute at least one action. In the sequel we consider the system can start in any configuration. That is, the local state of a node can be corrupted. We don't make any assumption on the number 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. The definition hardly uses the legitimate predicate. A legitimate predicate is defined over the configurations of a system and describes the set of correct configurations.

[1]The fined-grained communication atomicity model [7, 6] can be used to design more easily a self-stabilizing algorithm for message passing model. Each node maintains a local copy of the variables of its neighbors. These variables are refreshed via special messages exchanged periodically by neighboring nodes. Therefore, in the message passing model the space complexity of our algorithm is $O(\Delta \log^2 n)$ bits per node by considering also the local copies of neighbors' variables, with $\Delta$ the maximum degree of a node in the network.

**Definition 1 (self-stabilization)** *Let $\mathcal{L}_{\mathcal{A}}$ be a non-empty* legitimate predicate *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 $\mathcal{L}_{\mathcal{A}}$ and verifies Spec (*closure*).*
*(ii) Every computation of $\mathcal{A}$ starting from an arbitrary configuration contains a configuration that satisfies $\mathcal{L}_{\mathcal{A}}$ (*convergence*).*

To compute the time complexity, we use the definition of *round* [6]. Given a computation $e$ ($e \in \mathcal{E}$), the *first round* of $e$ (let us call it $e'$) is the minimal prefix of $e$ containing the execution of one action (an action of the protocol or a disabling action) of every enabled processor from the initial configuration. Let $e''$ be the suffix of $e$ such that $e = e'e''$. The *second round* of $e$ is the first round of $e''$.

## 2.2 Overview of our solution

We propose to extend the Gallager, Humblet and Spira (GHS) algorithm [9], to self-stabilizing settings via a compact informative labeling scheme. Thus, the resulting solution presents several advantages appealing to large scale systems: it is compact since it uses only memory whose size is poly-logarithmic in the size of the network, it scales well since it does not rely on any global parameter of the system.

The notion of a *fragment* is central to the GHS approach. A fragment is a sub-tree of the graph, i.e., a fragment is a tree which spans a subset of nodes. Note that a fragment can be limited to a single node. An outgoing edge of a fragment $F$ is an edge with a single endpoint in $F$. The minimum-weight outgoing edge of a fragment $F$ is an outgoing edge of $F$ with minimum weight among outgoing edges of $F$, denoted in the following as $\texttt{ME}_F$. In the GHS construction, initially each node is a fragment. For each fragment $F$, the GHS algorithm in [9] identifies the $\texttt{ME}_F$ and merges the two fragments endpoints of $\texttt{ME}_F$. It is important to mention that with this scheme, more than two fragments may be merged concurrently. The merging process is repeated in a iterative fashion until a single fragment remains. The result is a MST. The above approach is often called *blue rule* for MST construction [23].

This approach is particularly appealing when transient faults create a forest of fragments (which are sub-trees of a MST). The direct application of the blue rule allows the system to reconstruct a MST and to recover from faults which have divided the existing MST. However, when more severe faults hit the system the process' states may be corrupted leading to a configuration of the network where the set of fragments are not sub-trees of some MST. This may include, a spanning tree but not a MST or spanning structure containing cycles. In these different types of spanning structures, the application of the *blue rule* is not always sufficient to reconstruct a MST. To overcome this difficulty, we combine the *blue rule* with another method, referred in the literature as the *red rule* [23]. The *red rule* considers all the possible cycles in a graph, and removes the heaviest edge from every cycle, the resulting is a MST. To maintain a MST regardless of the starting configuration, we use the *red rule* as follows. Let $\texttt{T}$ denote a spanning tree of graph $G$, and $e$ an edge in $G$ but not in $\texttt{T}$. Clearly, if $e$ is added to $\texttt{T}$, this creates a (unique) cycle composed by $e$ and some edges of $\texttt{T}$. This cycle is called a *fundamental cycle*, and denoted by $C_e$. According to the *red rule*, if $e$ is not the edge of maximum weight in $C_e$, then there exists an edge $f \neq e$ in $C_e$, $f \in T$ such that $w(f) > w(e)$. In this case, $f$ can be removed since it is not part of any MST.

Our solution, called in the following `SS-MST` Algorithm, combines both the *blue rule* and *red rule*. The application of the *blue rule* needs that each node identifies the fragment it belongs to.

The *red rule* also requires that each node can identify the fundamental cycle associated to each of its adjacent non-tree-edges. Note that a simple scheme broadcasting the root identifier in each fragment (of memory size $O(\log n)$ bits per node) can be used to identify the fragments, but this cannot allow to identify fundamental cycles. In order to identify fragments or fundamental cycles, we use a self-stabilizing labeling scheme, called `NCA-L`. This scheme provides at each node a distinct label. For two nodes $u$ and $v$ in the same fragment, the comparison of their labels provides to these two nodes their **nearest common ancestor** in a tree (see Section 3). Thus, the advantage of this labeling is twofold. First the labeling scheme helps each node to identify the fragment it belongs to. Second, given any non-tree edge $e = \{u, v\}$, the path in the tree going from $u$ to the nearest common ancestor of $u$ and $v$, then from there to $v$, and finally back to $u$ by traversing $e$, constitute the fundamental cycle $C_e$.

To summarize, `SS-MST` algorithm will use the *blue rule* to build a spanning tree, and the *red rule* to recover from invalid configurations. In both cases, it uses our algorithm `NCA-L` to identify both fragments and fundamental cycles. Note that, in [18, 19] distributed algorithms using the *blue* and *red rules* to construct a MST in a dynamic network are proposed, however these algorithms are not self-stabilizing.

### 2.3 Notations

In this section we fix some general assumptions used in the current paper. Let $G = \langle V, E, w\rangle$ be an undirected weighted graph, where $V$ is the set of nodes, $E$ is the set of edges and the weight of each edge is given by a positive cost function $w : E \to \mathbb{R}^+$. We consider w.l.o.g. that the edges' weight are polynomial in $|V|$. Moreover, the nodes are allowed to have unique identifiers denoted by $\mathsf{Id}$ encoded using $O(\log n)$ bits where $n = |V|$. No assumption is made about the fact that edges' weight must be distinct. In the current paper $\mathsf{N}_v$ denotes the set of all neighbors of $v$ in $G$, for any node $v \in V(G)$.

Each node $v$ maintains several information a pointer to one of its neighbor node called *the parent*. The set of these pointers induces a spanning tree if the spanning structure is composed with all the nodes and contains no cycle. We denote by $\mathtt{path}(u, v)$ the path from $u$ to $v$ in the tree. For handling the nearest common ancestor labeling scheme we will define some notations. Let $\ell_v$ be the label of a node $v$ composed by a list of pairs of integers, where each pair is an identifier and a distance. $\ell_v[i]$ denotes the $i$th pair of the list, and for every pair $i$ the first element is denoted by $\ell_v[i][0]$ and the second one by $\ell_v[i][1]$. The last pair of the list is denoted by $\ell_v^{-1}$.

## 3 Self-stabilizing Nearest Common Ancestor Labeling scheme

Previously, we explained that our `SS-MST` algorithm needs to identify fragments, internal and outgoing edges of each fragment and the presence of cycles. To achieve this identification we use a nearest common ancestor labeling scheme. This section is dedicated to the presentation of a self-stabilizing version of the distributed algorithm proposed by Peleg [20]. The self-stabilizing algorithm is called in the following `NCA-L`. `NCA-L` algorithm can be used to solve other tasks than constructing a MST, hence we present this part in a separate section.

In [20], Peleg gives a nearest common ancestor labeling scheme for a tree structure with a memory complexity of $\Theta(\log^2 n)$ bits. We will first give in this section a self-stabilizing version of this scheme, that is the encoder and decoder part related to the labeling scheme, and finally we prove the correctness and the complexity of our self-stabilizing algorithm. For simplicity, we assume in this current section that all the nodes of the network belong to a single tree. It is easy to see that without a tree structure, the nodes cannot have a common ancestor. Therefore,

$$
\begin{array}{lcl}
\mathrm{C}(v) & = & \{u \in \mathsf{N}_v : p_u = \mathsf{Id}_v\} \\
\mathrm{nbrNdS}(v) & = & \Big(1 + \sum_{u \in \mathrm{C}(v)} size_u[0], \quad \max\{\mathsf{Id}_u : u \in \mathrm{C}(v) \wedge size_u[0] = \max\{size_x[0] : x \in \mathrm{C}(v)\}\}\Big) \\
\mathrm{Leaf}(v) & \equiv & (\mathrm{C}(v) = \emptyset \wedge size_v = (1, \perp)) \\
\mathrm{SizeC}(v) & \equiv & \mathrm{Leaf}(v) \vee (\mathrm{C}(v) \neq \emptyset \wedge size_v = \mathrm{nbrNdS}(v)) \\
\mathrm{Label}_R(v) & \equiv & (p_v = \emptyset \wedge \ell_v = (\mathsf{Id}_v, 0)) \\
\mathrm{Label}_{Nd}(v) & \equiv & (p_v \in N(v)) \wedge (\mathrm{Heavy}(v) \vee \mathrm{Light}(v)) \\
\mathrm{Label}(v) & \equiv & \mathrm{Label}_R(v) \vee \mathrm{Label}_{Nd}(v) \\
\mathrm{Heavy}(v) & \equiv & (size_{p_v}[1] = \mathsf{Id}_v) \wedge (size_v[0] < size_{p_v}[0]) \wedge (\ell_{p_v} \backslash \ell_{p_v}^{-1} = \ell_v \backslash \ell_v^{-1}) \wedge (\ell_{p_v}^{-1}[1] + 1 = \ell_v^{-1}[1]) \\
\mathrm{Light}(v) & \equiv & (size_{p_v}[1] \neq \mathsf{Id}_v) \wedge (size_v[0] \leq size_{p_v}[0]/2) \wedge (\ell_v = \ell_{p_v}.(\mathsf{Id}_v, 0))
\end{array}
$$

Figure 1: Macros and predicates of Algorithm `NCA-L` for any $v \in V$.

in the next section we have to deal with the general case in which cycles can be contained in the starting configuration.

### 3.1 Variables

Before presenting the nearest common ancestor labeling scheme, we describe below the variables used by the labeling scheme. Each node $v \in V$ maintains three variables:

- A parent pointer to a neighbor of $v$ stored in $p_v$ defining the spanning tree.
- $size_v$ is a pair of integers, whose the first element is an estimation of the number of nodes in the sub-tree of $v$ and the second one is the identifier of the child of $v$ with the subtree of highest size. If $v$ has no child then $size_v$ is equal to $(1, \perp)$. Note that, the first integer of the pair is referenced by $size_v[0]$, while the second integer by $size_v[1]$.
- The label of $v$ (composed of a list of pairs of integers where each pair is an identifier and a distance (described below)) is stored in variable $\ell_v$.

We will now present the manner the nearest common ancestor scheme computes the label of each node in a spanning tree.

### 3.2 Labeling encoder

The main idea of this protocol is to divide a tree structure in sub-paths to minimize the label size of each node. Let us describe more precisely our self-stabilizing version of this protocol. In a rooted tree, a *heavy* edge is an edge between a node $u$ and one of its children $v$ with the highest number of nodes in its sub-tree. The other edges between $u$ and its other children are tagged as *light* edges. We extend this edge designation to the nodes, a node $v$ is called *heavy node* if the edge between $v$ and its parent is a heavy edge (see Predicate Heavy($v$) in Figure 1), otherwise $v$ is called *light node* (see Predicate Light($v$) in Figure 1). Moreover, the root of a tree is a heavy node. The idea of the scheme is as follows. A tree is recursively divided into paths of disjoint edges: *heavy* and *light* paths. Remark: Any child of highest number of nodes can be selected as heavy node, so among these children the one of highest identifier can be selected.

To label the nodes in a tree $\mathsf{T}$, the size of each subtree rooted at each node of $\mathsf{T}$ is needed to identify heavy edges leading the heaviest subtrees at each level of $\mathsf{T}$. To this end, each node $v$ maintains a variable named $size_v$ which is a pair of integers. The first integer is the local estimation of the number of nodes in the subtree rooted at $v$. For a node $v$ this value is computed by summing up all the estimated values of its children plus one. The value of

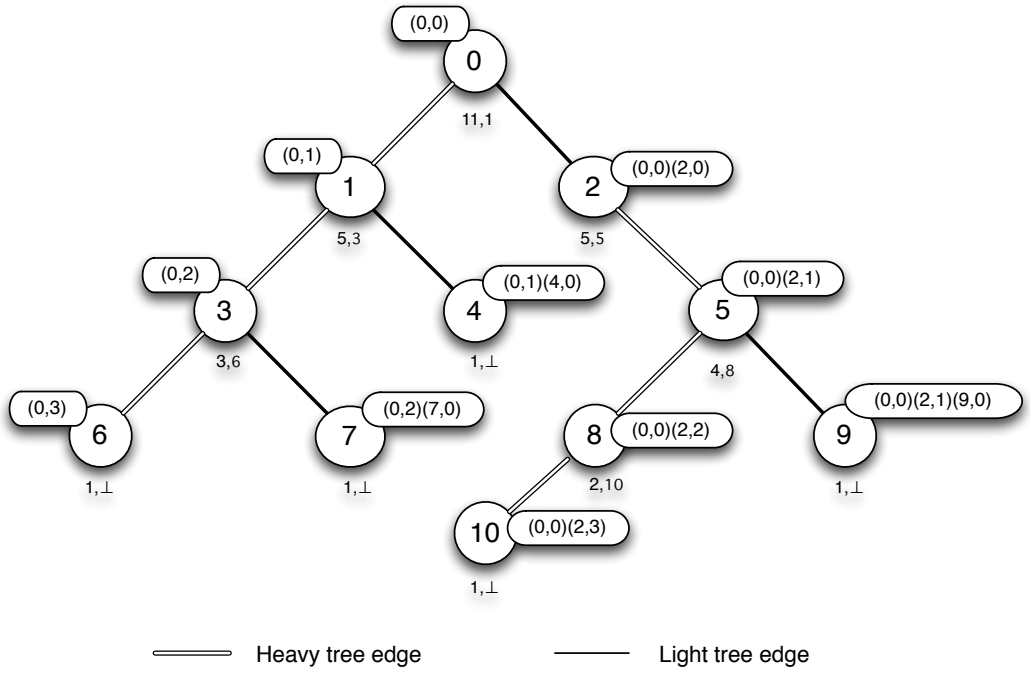


Figure 2: Nearest Common Ancestor Labeling scheme for a tree. The bubble at each node $v$ corresponds to the label of $v$. The integer inside each node corresponds to the node's identifier, while the other notation corresponds to the variable *size*.

$size_v$ is processed in a bottom-up fashion from the leaves to the root of the tree (see Predicate SizeC($v$) in Figure 1 and rule $\mathrm{R}_{\texttt{Size}}$). The second integer is the identifier of a child of $v$ with maximum number of nodes in its sub-tree, which indicates the heavy edge. We suppose w.l.o.g that, in case of equality between the size of the children's subtrees the child with the minimum identity is chosen. The variable $size_v$ is setted to $(1, \perp)$ for a leaf node $v$ (see Predicate Leaf($v$) in Figure 1).

Based on the heavy and light nodes in a tree $\mathsf{T}$ indicated by variable $size_v$ at each node $v \in \mathsf{T}$, each node of $\mathsf{T}$ can compute its label (see rule $\mathrm{R}_{\texttt{Label}}$ in Figure 3). The label of a node $v$ stored in $\ell_v$ is a list of pair of integers. Each pair of the list contains the identifier of the node which is the root of the heavy path (i.e., a path including only heavy edges) that $v$ belongs to and the distance to it. For the root $v$ of a fragment, the label $\ell_v$ is the following pair $(\mathsf{Id}_v, 0)$, respectively the identifier of $v$ and the distance to itself, i.e., zero (see Predicate $\mathrm{Label}_R(v)$ in Figure 1). When a node $u$ is tagged by its parent as a heavy node (i.e., $size_{p_v}[1] = \mathsf{Id}_u$), then the node $u$ takes the label of its parent but it increases by one the distance of the last pair of the parent label (see Predicate $\mathrm{Label}_{Nd}(v)$ in Figure 1).
Otherwise, a node $u$ is tagged by its parent $v$ as a light node (i.e., $size_{p_v}[1] \neq \mathsf{Id}_u$), then the node $u$ becomes the root of a heavy path and it takes the following label: the label of its parent to which $u$ concatenates to a new pair composed by its identifier and a zero distance (we note the step of concatenation by the operator ".").
Examples of theses cases are given in Figure 2, where integers inside the nodes are node identifiers and lists of pairs of values are node labels.

Algorithm `NCA-L` is composed by the rules $\mathrm{R}_{\texttt{Size}}$ and $\mathrm{R}_{\texttt{Label}}$ given in Figure 3 which correct the variables *size* and $\ell$ respectively if needed.

$R_{\texttt{Size}}$: [ **Size correction** ]

**If** $\neg\text{SizeC}(v)$ **Then**
  **If** $\text{C}(v) = \emptyset$ **then** $size_v := (1, \perp)$
  **Else** $size_v := \text{nbrNdS}(v)$;

$R_{\texttt{Label}}$: [ **Label correction** ]

**If** $\text{SizeC}(v) \wedge \neg\text{Label}(v)$ **Then**
  **If** $size_{p_v}[1] = \mathsf{Id}_v$ **then** $\ell_v := \ell_{p_v}; \ell_v^{-1}[1] := \ell_v^{-1}[1] + 1;$
  **Else** $\ell_v := \ell_{p_v}.(\mathsf{Id}_v, 0)$

Figure 3: Formal description of Algorithm `NCA-L` for any $v \in V$.

### 3.3 Labeling decoder

Let us now describe the decoder for the nearest common ancestor. This decoder is given in [20], but for simplicity we present it using our own notations (see predicate *nca* in Figure 4). Let us consider two nodes $u$ and $v$, we denote by $nca(\ell_u, \ell_v)$ the label of the nearest common ancestor of $u$ and $v$. For the remainder of this paper, we define the following notations: $\ell^{\cap}_{u,v} = \ell_u \cap \ell_v$ and $\ell'_{u,v} = \ell_u \backslash \ell^{\cap}_{u,v}$. The nearest common ancestor of $u$ and $v$ is composed by the common part of the label of $u$ and $v$ ($\ell^{\cap}_{u,v}$) and by the smaller pair following the lexicographic order of the last pair of their labels (i.e., minimum between $\ell'_u[0]$ and $\ell'_v[0]$). In the other case $u$ and $v$ have not common ancestor.

$$nca(\ell_u, \ell_v) \equiv \begin{cases} \ell^{\cap}_{u,v}.\ell'_{u,v}[0] & \textbf{If } \ell'_{u,v}[0][0] = \ell'_{v,u}[0][0] \wedge \ell'_{u,v}[0][1] < \ell'_{v,u}[0][1] \wedge \ell^{\cap}_{u,v} \neq \emptyset \\ \ell^{\cap}_{u,v}.\ell'_{v,u}[0] & \textbf{If } \ell'_{u,v}[0][0] = \ell'_{v,u}[0][0] \wedge \ell'_{v,u}[0][1] > \ell'_{u,v}[0][1] \wedge \ell^{\cap}_{u,v} \neq \emptyset \\ \emptyset & \textbf{otherwise} \end{cases}$$

Figure 4: Macro used for computing the nearest common ancestor.

On the example defined on Figure 2, the labels of nodes 9 and 10 are respectively $\ell_9 = (0,0)(2,1)(9,0)$ and $\ell_{10} = (0,0)(2,3)$. In this case, we have for the defined notations on labels: $\ell^{\cap}_{9,10} = (0,0)$, $\ell'_{10,9} = (2,1)(9,0)$ and $\ell'_{9,10} = (2,3)$. Since we have $\ell'_{9,10}[0][1] < \ell'_{10,9}[0][1]$ then on this example $nca(\ell_9, \ell_{10}) = (0,0)(2,1)$.

### 3.4 Correctness and complexity

This subsection is dedicated to the correctness of the self-stabilizing nearest common ancestor labeling scheme. Let $\Gamma$ be the set of all possible configurations of the system. In order to prove the correctness of the `NCA-L` algorithm, we denote $\Gamma_{size}$ the set of configurations in $\Gamma$ such that variables *size* are correct in the system. More precisely, we define the following function $s$: $V \to \mathbb{N}$ be the function defined by

$$s(v) = |\big((size_v[0] - 1) - \sum_{u \in \mathcal{C}(v)} size_u[0]\big)|.$$

Note that $s(v) \geq 0$, and the variable *size* has a correct value at node $v$ if and only if $s(v) = 0$. In the following, we show that any execution of the system converges to a configuration in $\Gamma_{size}$, and the set of configurations $\Gamma_{size}$ is closed. The following lemma establishes the former property. We assume that all the nodes of the system belongs to the tree `T` and we define below a legitimate configuration for the informative labeling scheme considered in this section.

**Definition 2 (Legitimate configuration for Labeling scheme)** *A configuration $\gamma \in \Gamma_\Lambda$ is called legitimate if the following conditions are satisfied:*

1. *the root node $r$ of the tree $\mathtt{T}$ has label equal to $(\mathsf{Id}_r, 0)$,*
2. *every* heavy *node $v \in \mathtt{T}$ has a label equal to $(\ell_{p_v} \backslash \ell^{-1}_{p_v}).(\ell^{-1}_{p_v}[0], \ell^{-1}_{p_v}[1] + 1)$,*
3. *every* light *node $v \in \mathtt{T}$ has a label equal to $\ell_{p_v}.(\mathsf{Id}_v, 0)$.*

**Lemma 1** *Starting from an arbitrary configuration $\gamma \in \Gamma$, the system reaches a configuration in $\gamma' \in \Gamma_{size}$ in $O(\delta_\mathtt{T})$ rounds, where $\delta_\mathtt{T}$ is the depth of the tree $\mathtt{T}$.*

**Proof.** First, we define the following potential function $\Phi$. We denote by $\delta_\mathtt{T}$ the depth of the tree $\mathtt{T}$, i.e., the length of the longest path from the root to the leaves. Let $\gamma \in \Gamma$ be a configuration, and let $\Gamma$ be the set of all configurations. Let $\Phi : \Gamma \to \mathbb{N}$ be the function defined by

$$\Phi(\gamma) = \sum_{d=0}^{\delta_T} \nu_d(\gamma)(n+1)^d$$

where $\nu_d(\gamma)$ is the number of nodes $v$ at depth $d$ in $\mathtt{T}$ with $s(v) \neq 0$. Note that $0 \leq \nu_d(\gamma) \leq n$, and $0 \leq \Phi(\gamma) \leq (n+1)^{\delta_T+1}$. Also, the variable *size* has a correct value at every node if and only if $\Phi(\gamma) = 0$. Let $\gamma(t)$ denotes the configuration of the system after round $t$. Let $d_0$ be the largest index such that $\nu_{d_0}(\gamma(t)) \neq 0$. Since we use a weakly fair scheduler, all the nodes are scheduled during the execution of round $t+1$. Every node $v$ at depth $d > d_0$ does not change its value of variable *size* (see the predicate SizeC), and therefore $s(v)$ remains zero, so $\nu_d(\gamma(t+1))$ remains zero as well. The nodes at depth $d_0$ change their variable *size* according to the variable *size* of their children. Let $v$ be a node at depth $d_0$. The children of $v$ (if any) are at depth $d > d_0$. Thus, their variable *size* has not changed, and therefore $s(v)$ becomes zero after round $t+1$. As a consequence, $\nu_{d_0}(\gamma(t+1)) = 0$. Therefore, we get

$$\Phi(\gamma(t+1)) < \Phi(\gamma(t))$$

and thus the system will eventually reach a configuration in $\Gamma_{size}$. To measure the number of rounds it takes to get into $\Gamma_{size}$, observe that $\delta_\mathtt{T}$ decreases by at least one at each round.Starting from any arbitrary configuration, the system reaches a configuration in $\Gamma_{size}$ in $O(\delta_\mathtt{T})$ rounds. □

**Lemma 2** *Starting from a configuration in $\Gamma_{size}$ the system can only reach configurations in $\Gamma_{size}$.*

**Proof.** According to algorithm `NCA-L`, the variable *size* is modified only by Rule $\mathrm{R}_{\mathtt{Size}}$. Consider a configuration $\gamma \in \Gamma_{size}$ such that variables *size* are correct. For each node $v$, we have $s(v) = 0$ and Predicate SizeC($v$) is true. Thus, Rule $\mathrm{R}_{\mathtt{Size}}$ cannot be executed by a node $v$ and we have $s(v) = 0$ which implied that $\Phi(\gamma) = 0$. Therefore, for any execution starting from a configuration $\gamma \in \Gamma_{size}$, the system remains in a configuration in $\Gamma_{size}$. □

**Lemma 3 (Convergence for `NCA-L`)** *Starting from an illegitimate configuration, Algorithm* `NCA-L` *reaches in $O(\delta_\mathtt{T})$ rounds a legitimate configuration, where $\delta_\mathtt{T}$ is the depth of the tree $\mathtt{T}$.*

**Proof.** Let us introduce some notations that we will use throughout in the proof. Let $\bar{\ell}_v = \ell_v \backslash \ell_v^{-1}$ be the pairs list of the node's label $v$ such that the last pair is removed, and $|\ell_v|$ the number of pairs in the label of $v$. For two labels $\ell_v$ and $\ell_u$ the step $\ominus$ is defined by:

$$\ell_v \ominus \ell_u = \sum_{i=0}^{|\ell_v|-1} |\ell_v[i][0] - \ell_u[i][0]| + |\ell_v[i][1] - \ell_u[i][1]|.$$

We first define a first function $L(v)$ on the state of each node $v \in V$ as following:

$$L(v) \equiv \begin{cases} s(v) + ||\ell_v| - 1| + \ell_v \ominus (\mathsf{Id}_v, 0) & \text{If } v = r \\ s(v) + ||\ell_v| - |\ell_{p_v}|| + (\bar{\ell}_v \ominus \bar{\ell}_{p_v}) + |\ell_v^{-1}[0] - \ell_{p_v}^{-1}[0]| + |\ell_v^{-1}[1] - \ell_{p_v}^{-1}[1] - 1| & \text{If } size_{p_v}[1] = \mathsf{Id}_v \\ s(v) + ||\ell_v| - |\ell_{p_v}| - 1| + (\bar{\ell}_v \ominus \ell_{p_v}) + (\ell_v^{-1} \ominus (\mathsf{Id}_v, 0)) & \text{Otherwise} \end{cases}$$

Note that $L(v) \geq 0$ and when $L(v) = 0$ the variable $\ell$ has a correct value for a node $v$. Let $\Lambda$: $\Gamma \to \mathbb{N}$ be the function defined by,

$$\Lambda(\gamma) = \sum_{d=0}^{\delta_T} \xi_d(\gamma)(n+1)^{n+1-d}$$

where $\xi_d(\gamma)$ is the number of nodes $v$ at depth $d$ in $\mathtt{T}$ with $L(v) \neq 0$. Remark that $0 \leq \xi_d(\gamma) \leq n$, and $0 \leq \Lambda(\gamma)$. Also, the variable $\ell$ has a correct value at every node if and only if $\Lambda(\gamma) = 0$. Let $\gamma(t)$ denote the configuration of the system after round $t$, and suppose that $t > n$. By lemma 1 and lemma 2 we prove that $\gamma(t) \in \Gamma_{size}$. It is important to mention that, in $\gamma(t)$ all node $v$ can check if it is a heavy node or light node (see variable *size*). Let $d_0$ be the smallest index such that $\xi_{d_0}(\gamma(t)) \neq 0$. Since we use a weakly fair scheduler, all the nodes are scheduled during the execution of round $t+1$. Every node $v$ at depth $d < d_0$ does not change its value of variable $\ell$ (see the predicate Label), and therefore $L(v)$ remains zero, so $\xi_d(\gamma(t+1))$ remains zero as well. The nodes at depth $d_0$ change their variable $\ell$ according to the variables $\ell$ and *size*[1] of their parent (see Rule $\mathrm{R}_{\mathtt{Label}}$). Let $v$ be a node at depth $d_0$. The parent of $v$ is at depth $d < d_0$. Thus, its variable $\ell$ have not changed, and therefore $L(v)$ becomes zero after round $t+1$. As a consequence, $\xi_{d_0}(\gamma(t+1)) = 0$. Therefore, we get

$$\Lambda(\gamma(t+1)) < \Lambda(\gamma(t))$$

and thus the system will eventually reach a legitimate configuration for algorithm `NCA-L`. To measure the number of rounds it takes to get into a legitimate configuration for algorithm `NCA-L`, observe that $\delta_T$ decreases by at least one at each round. Since $\delta_T \leq n-1$ for every $\gamma \in \Gamma_{size}$, we get that, starting from any configuration in $\Gamma_{size}$ configuration, the system reaches a legitimate configuration for algorithm `NCA-L` in $O(\delta_{\mathtt{T}})$ rounds. Using lemma 1 and lemma 2, we can conclude starting from any arbitrary configuration, the system reaches a legitimate configuration for algorithm `NCA-L` in $O(\delta_{\mathtt{T}})$ rounds. □

**Lemma 4 (Closure for `NCA-L`)** *The set of legitimate configurations for* `NCA-L` *is closed. That is, starting from any legitimate configuration, the system remains in a legitimate configuration.*

**Proof.** According to Algorithm `NCA-L`, the labeling procedure is done using only Rule $\mathrm{R}_{\mathtt{Label}}$. Let $\gamma$ a legitimate configuration. For each node $v$ in $\gamma$, we have $\Phi(\gamma) = 0$ and $\Lambda(\gamma) = 0$. Moreover in $\gamma$, Predicates SizeC($v$) and Label($v$) are true and Rules $\mathrm{R}_{\mathtt{Label}}$ and $\mathrm{R}_{\mathtt{Size}}$ cannot be executed by any node $v \in V$. In conclusion, starting from a legitimate configuration for algorithm `NCA-L` the system remains in a legitimate configuration. □

The following theorem is a direct consequence from Lemmas 3 and 4.

**Theorem 1** *Algorithm* `NCA-L` *is self-stabilizing for the informative nearest common ancestor labeling scheme.*

# 4 Self-Stabilizing Minimum Spanning Tree Algorithm

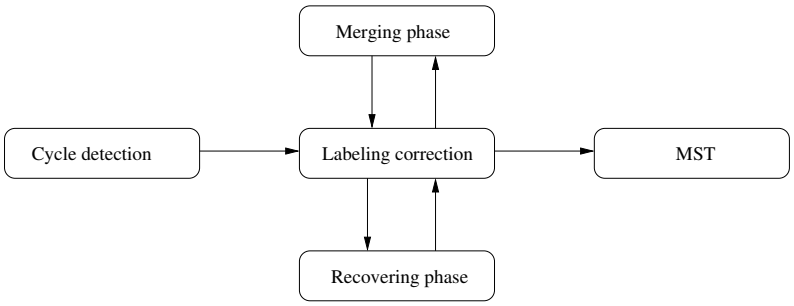


Figure 5: Relation between the different phases of Algorithm `SS-MST`.

In this section we describe our self-stabilizing algorithm for constructing the minimum spanning tree, called `SS-MST` algorithm. Our `SS-MST` algorithm uses the "blue rule" to construct a spanning tree and the "red rule" to recover from invalid configurations (see section 2.2). In both cases, it uses `NCA-L` algorithm to identify fragments and fundamental cycles. We assume in the following that the *merging* phases have a higher priority than the *recovering* phases. That is, the system recovers from an invalid configuration if and only if no merging is possible.

Unfortunately, due to arbitrary initial configuration, the structure induced by the parent pointer of all nodes may contain cycles. We use first a well known approach to break cycles before giving a detailed description of merging and recovering phases.

Figure 5 illustrates the different phases of Algorithm `SS-MST`. Starting from an arbitrary configuration, first all the cycles are destroyed then fragments are defined and correctly labeled using the parent pointers. Based on the label of nodes, the minimum *outgoing edge* (i.e., edge whose extremities belong to different fragments) of each fragment is computed in a bottom-up fashion, and allowing to a pair of fragments which have selected the same outgoing edge to be merged together through this edge. A *merging step* gives a new fragment which is the result of the merging of a pair of fragments. When a new fragment is created, the nodes of this fragment have to compute their new label. This process is repeated until there is only one remaining fragment spanning all the nodes of the network. In this case, the recovering phase can begin by detecting that no outgoing edge can be selected. To handle this phase each fragment has to compute its *internal edges* (i.e., edges whose extremities belong to the same fragment) and to identify the *nearest common ancestor* based on the labels of the edge extremities. The weight of the internal edges are broadcasted up in the tree from the leaves to the root. Let $e = \{u, v\}$ an internal edge of Tree `T`, due to the "red rule" if an edge $f$ of the path $\texttt{path}(u, nca(\ell_u, \ell_v))$ in `T` has a weight bigger than $e$, then $e$ is an *valid edge* since $e$ is part of an `MST` (by "red rule"). More precisely, if during the bottom-up transmission of the weight of $e$, a node $u$ has a parent link edge $f$ such that $w(f) > w(e)$ then $f$ is deleted from the tree `T` and $u$ becomes the root of a new fragment.

We present first the variables used by Algorithm `SS-MST`, then we describe the approach used to delete the cycles, followed by the merging and recovering phases. Finally, we show the correctness and the time and memory complexities of the algorithm.

### 4.1 Variables

We list below the eight variables maintained at each node $v \in V$:

- The three variables described in Section 3 are used, i.e., variables $p_v, size_v$ and $\ell_v$.
- The distance of each node $v$ from the root of the fragment is stored in variable $d_v$.
- For handling the *blue rule* mentioned in section 2.2, the minimum outgoing edge of each fragment is stored in Variable $Out_v$. This edge is composed of three elements: the edge weight, and the identifiers of the edge extremities. The $i$-th element of $Out_v$ is accessed by $Out_v[i]$ with $i \in \{0, ..., 2\}$.
- Finally to broadcast the internal edges in the recovering phase, a last variable $In_v$ stores three elements related to an internal edge: the edge weight, and the labels of the edge extremities. As for Variable $Out_v$, the $i$-th element of $In_v$ is accessed by $In_v[i]$ with $i \in \{0, ..., 2\}$.

### 4.2 Cycles detection and Labels correction

The previous section was dedicated to the labeling procedure for an unique tree, due to the arbitrary starting configuration, the network can contain a forest of subtrees (several fragments) and cycles. Therefore, the labeling procedure described in previous section (using Rules $\mathrm{R}_{\texttt{Size}}$ and $\mathrm{R}_{\texttt{Label}}$) is executed separately in each subtree in Algorithm `SS-MST`. However, to apply this procedure it is crucial to detect the cycles in the fragments induced by the parent pointers. To this end, we use a common approach used to break cycles in a spanning structure [8]. Each node computes its distance (in hops) to the root by using the distance of its parent plus one. By following this procedure, there is at least a node which has a distance higher or equal than the distance of its parent in the fragment. Therefore, this condition is used at each node to detect a cycle. In this case, a node $v$ deletes its parent pointer by selecting no parent and a new fragment rooted at $v$ is created. Unfortunately, due to the arbitrary initial configuration a cycle can be falsely detected because of erroneous distances values at $v$ and its parent. This mechanism based on distances ensures that after $O(n)$ rounds the network is cycle free. The destruction of cycles is managed by rule $\mathrm{R}_{\texttt{Correct}}$.

When all the cycles have been deleted, the labeling procedure is applied in Algorithm `SS-MST`. Note that the cycle detection must have a higher priority over the labeling procedure. To this end, Rule $\mathrm{R}_{\texttt{Correct}}$ is the first rule to execute and in exclusion with Rules $\mathrm{R}_{\texttt{Size}}$ and $\mathrm{R}_{\texttt{Label}}$ in Algorithm `SS-MST`. Furthermore, the labeling scheme must also have a higher priority over the merging and recovering phases. Indeed, the label of the nodes are used to identify the internal and outgoing edges of a fragment (see Figure 7). To guarantee the execution priority, the rules of the labeling scheme can only be executed when Predicate $Distance(v)$ is satisfied at node $v$. In the same way, the rules of merging and recovering phases can only be executed at a node $v$ when Predicate $CorrectF(v)$ is satisfied at $v$.

$$
\begin{array}{lcl}
\mathrm{Distance}(v) & \equiv & (p_v = \emptyset \wedge d_v = 0) \vee (p_v \neq \emptyset \wedge d_v = d_{p_v} + 1) \\
\mathrm{SizeC}(v) & \equiv & \mathrm{Leaf}(v) \vee (\mathrm{C}(v) \neq \emptyset \wedge size_v = \mathrm{nbrNdS}(v)) \\
\mathrm{Label}(v) & \equiv & \mathrm{Label}_R(v) \vee \mathrm{Label}_{Nd}(v) \\
CorrectF(v) & \equiv & \mathrm{Distance}(v) \wedge \mathrm{SizeC}(v) \wedge \mathrm{Label}(v)
\end{array}
$$

Figure 6: Predicates used by Rule $\mathrm{R}_{\texttt{Correct}}$ and labeling rules.

---

$R_{\texttt{Correct}}$: [ **Correction** ]

**If** $\neg$Distance($v$) **Then**
$Out_v = \emptyset; In_v = \emptyset$
**If** $(p_v = \emptyset) \wedge d_v \neq 0$ **Then** $d_v := 0$;
**If** $(p_v \neq \emptyset) \wedge (d_{p_v} + 1 < d_v)$ **Then** $d_v := d_{p_v} + 1$;
**If** $(p_v \neq \emptyset) \wedge (d_{p_v} \geq d_v)$ **Then** $p_v := \emptyset$; $\ell_v := (\mathsf{Id}_v, 0); d_v := 0$;

---

We give below the rules associated with the labeling encoder (given in the previous section). In order to use these two rules for the MST construction, we add Predicate Distance($v$) in the guards. This allow to disable these rules when a cycle is detected with Rule $R_{\texttt{Correct}}$.

---

$R_{\texttt{Size}}$: [ **Size correction** ]

**If** Distance($v$) $\wedge$ $\neg$SizeC($v$) **Then**
**If** C($v$) $= \emptyset$ **then** $size_v := (1, \perp)$
**Else** $size_v :=$ nbrNdS($v$);

$R_{\texttt{Label}}$: [ **Label correction** ]

**If** Distance($v$) $\wedge$ SizeC($v$) $\wedge$ $\neg$Label($v$) **Then**
**If** $size_{p_v}[1] = \mathsf{Id}_v$ **then** $\ell_v := \ell_{p_v}; \ell_v^{-1}[1] := \ell_v^{-1}[1] + 1$;
**Else** $\ell_v := \ell_{p_v}.(\mathsf{Id}_v, 0)$

---

### 4.3 Merging phase

When the graph induced by the parent pointers is cycle free and every node $v$ of a fragment $F$ has a correct label (see Predicate $CorrectF(v)$), then every node $v \in F$ is able to determine if $F$ spans all the nodes of the network or not. This knowledge is given by the label of the nodes, more precisely using the decoder given in Subsection 3.3. Indeed, given a non-tree edge $e = \{u, v\}$, if the nodes $u$ and $v$ have no common ancestor then $u$ and $v$ are in two distinct fragments. In this case, the merging phase can be executed at $u$ and $v$. A merging phase is composed of several *merging steps* in which at least two fragments are merged. Each merging step is performed following four steps:

1. The root of each fragment $F$ identifies the minimum-weight outgoing edge $e = (u, v)$ of its fragment (see Rule $R_{\texttt{Min}}$).

2. After the computation of $e$ each node $x$ on the path between the root of $F$ and $v \in F$ computes in variable $newp_x$ its future parent (see rule $R_{\texttt{Merge}}$). The nodes in the sub-tree rooted at every node $x$ executes also Rule $R_{\texttt{Merge}}$.

3. When the two merging fragments have finished the two first steps, then each node of these two fragments can compute their future distance (see Rule $R_{\texttt{Dist}}$).

4. Finally, every node $v$ belonging to these two fragments copies the content of its variables $newp_v$ (resp. $newd_v$) into variable $p_v$ (resp. $d_v$).

Let us proceed with a more detailed description of the these steps. We process the computation of the minimum-weight outgoing edge of each fragment in a bottom-up manner (see Rule $R_{\texttt{Min}}$). Each node $v$ can identify its adjacent outgoing edges $e = \{u, v\}$ by computing locally that $e$ has no nearest common ancestor using the labels of $u$ and $v$. This is done via the decoder given in subsection 3.3 and Macro OE($v$) at $v$. Each node $v$ computes the minimum-weight outgoing edge of its sub-tree (given by Macro Candidate($u$)) by selecting the edge of minimum-weight among its adjacent outgoing edges (given by Macro $\text{Cand}_l(v)$) and the one given by its children (given by Macro $\text{Cand}_c(v)$). The weight and the identifier of the extremities of the minimum-weight outgoing edge are stored in variable $Out_v$ at $v$. All these information will be used for the merging step. Figure 7 depicts the selection of the minimum outgoing edge for two fragments.

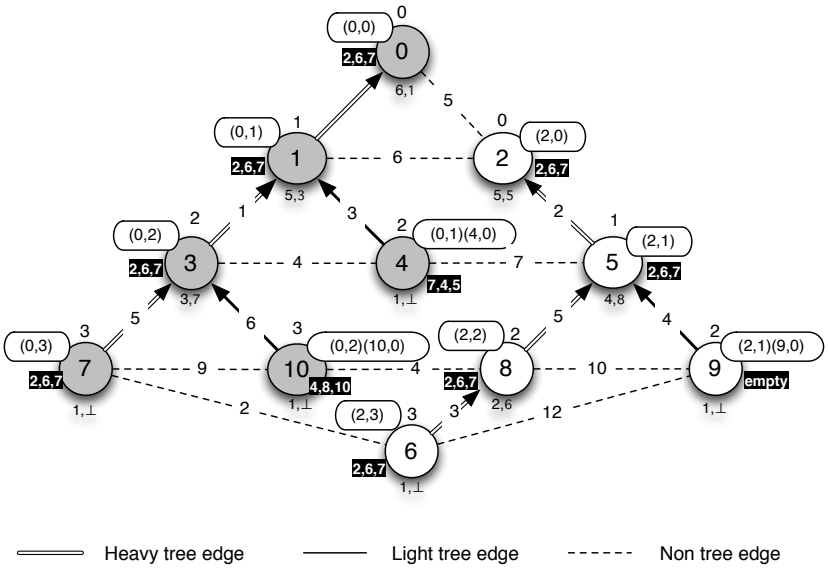


Figure 7: Minimum weight outgoing edge computation based on Nearest Common Ancestor Labeling scheme for a forest: The white bubble at each node $v$ corresponds to the label of the node. The black bubble at each node represent the selection of minimum outgoing edge. The information under the node corresponds to the variable *size* and the information on top of the node represent the distance of the node from the root.

When the computation of the minimum-weight outgoing edge $e = (u, v)$ is finished at the root $r$ of a fragment $F$ (i.e., $Out_r = \text{Candidate}(r)$), then $r$ can start the computation of the future parent pointers in $F$ (Predicate $ChangeNewP(r)$ is satisfied), done in a top-down manner (see rule $R_{\texttt{Merge}}$). Let $u$ be the extremity of $e$ of minimum identity between $u$ and $v$. If $e$ is selected as the minimum-weight outgoing edge of two fragments $F$ and $F'$, then $u$ will become the new root of Fragment $F''$ resulting from the merging between $F$ and $F'$. Otherwise, $e$ is the minimum-weight outgoing edge selected only by a single fragment, w.l.o.g. let $F$. In this case, $F$ will wait for that $e$ is selected as the minimum-weight outgoing edge of $F'$. In the two cases, every node $v$ of a fragment in a merging step computes its future parent pointer in variable $Out_v$. Each node on the path from the root of the fragment leading to the minimum-weight outgoing edge selects its child on this path as its future parent, while the other nodes select their actual parent.

$$
\begin{array}{lcl}
\mathrm{C}(v) & = & \{u \in \mathsf{N}_v : p_u = \mathsf{Id}_v\} \\
\mathrm{OE}(v) & = & \min\{(u,v) : u \in N_v \setminus (\mathrm{C}(v) \cup \{p_v\}) \wedge nca(\ell_u, \ell_v) = \emptyset\} \\
NCand(val) & = & \begin{cases} \min\{(u,v) \in \mathrm{OE}(v) : w(u,v) = val\} & \text{If } val = \mathrm{Cand}_c(v) \\ \min\{Out_u[1] : u \in \mathrm{C}(v) \wedge Out_u[0] = val\} & \text{Otherwise} \end{cases} \\
\mathrm{Cand}_c(v) & = & \min\{Out_u[0] : u \in \mathrm{C}(v)\} \\
\mathrm{Cand}_l(v) & = & \min\{w\{v,u\} : (u,v) \in \mathrm{OE}(v)\} \\
\mathrm{Candidate}(v) & = & \min\{\mathrm{Cand}_c(v), \mathrm{Cand}_l(v)\} \\
NewParent(v) & = & \begin{cases} \min\{u \in N_v : (u,v) = Out_v[1]\} & \text{If } Out_v[0] = \mathrm{Cand}_l(v) \wedge Out_v[1] \in \mathrm{OE}(v) \\ \min\{u \in \mathrm{C}(v) : Out_u = Out_v\} & \text{Otherwise} \end{cases} \\
NewDist(v) & = & \begin{cases} 0 & \text{If } newp_{newp_v} = v \wedge \mathsf{Id}_{newp_v} > \mathsf{Id}_v \\ 1 & \text{If } newp_{newp_v} = v \wedge \mathsf{Id}_{newp_v} < \mathsf{Id}_v \\ newd_{newp_v} + 1 & \text{Otherwise} \end{cases} \\
NewChild(v) & = & \{u : (u \in (C(v) \cup \{p_v\}) \wedge newp_u = v) \\
 & & \quad \vee (u \in N_v \wedge newp_u = v \wedge newp_v = u \wedge \mathsf{Id}_{newp_v} > \mathsf{Id}_v)\}
\end{array}
$$

$$
\begin{array}{lcl}
CorrectF(v) & \equiv & \mathrm{Distance}(v) \wedge \mathrm{SizeC}(v) \wedge \mathrm{Label}(v) \\
Reorientation(v) & \equiv & p_v = \emptyset \vee (Out_{p_v} = Out_v \wedge newp_{p_v} = v) \\
ChangeNewP(v) & \equiv & (Reorientation(v) \wedge newp_v \neq NewParent(v)) \vee (Out_{p_v} \neq Out_v \wedge newp_v \neq p_v) \\
ChangeNewD(v) & \equiv & (newp_{newp_v} = v \wedge newd_v > 1) \vee (p_{newp_v} = v \wedge newd_v \neq newd_{newp_v} + 1) \\
 & & \vee (newd_{p_v} \neq d_{p_v} \wedge newd_v \neq newd_{p_v} + 1) \\
CopyVar(v) & \equiv & (\forall u \in NewChild(v), d_u = newd_u \wedge p_u = newp_u) \\
 & & \wedge (p_v \neq newp_v \vee d_v \neq newd_v)
\end{array}
$$

Figure 8: Macros and predicates used by Algorithm `SS-MST` for the merging.

When $e$ is selected as the minimum-weight outgoing edge by $F$ and $F'$ and the computation of the future parent is done (i.e., $\neg ChangeNewP(v)$ is satisfied), then the future distance is computed in variable $newd_v$ by each node $v$ in $F \cup F'$ (Predicate $ChangeNewD(v)$ is satisfied), in a top-down manner following the parent relation given by variable $newp_v$ (see Rule $\mathrm{R}_{\texttt{Dist}}$). Note that the extremity of $e$ with the minimum identifier becomes the root of the new fragment with a zero distance. Finally, when the future parent and distance are computed by every node $v$ in $F \cup F'$ then $v$ can execute Rule $\mathrm{R}_{\texttt{End}}$ (see Predicate $CopyVar(v)$) to copy the content of variable $newp_v$ (resp. $newd_v$) into variable $p_v$ (resp. $d_v$). Note that this is done in a bottom-up fashion following the parent relation given by variable $newp_v$ in order to not destabilize Fragment $F$ or $F'$.

---

$\mathrm{R}_{\texttt{Min}}$: [ **Minimum computation** ]
**If** $CorrectF(v) \wedge \big(Out_v[0] \neq \mathrm{Candidate}(v) \neq \emptyset\big)$ **Then**
$Out_v := (\mathrm{Candidate}(v), NCand(\mathrm{Candidate}(v)))$;

$\mathrm{R}_{\texttt{Merge}}$: [ **Merging** ]
**If** $CorrectF(v) \wedge \big(Out_v[0] = \mathrm{Candidate}(v) \neq \emptyset\big) \wedge ChangeNewP(v)$ **Then**
$newd_v := \infty$;
**If** $Reorientation(v)$ **Then** $newp_v := NewParent(v)$;
**Else** $newp_v := p_v$

$R_{\texttt{Dist}}$: [ **New distance** ]
**If** $CorrectF(v) \wedge \neg ChangeNewP(v) \wedge ChangeNewD(v)$ **Then**
$newd_v := NewDist(v)$;

$R_{\texttt{End}}$: [ **End of merging** ]
**If** Distance$(v) \wedge \neg ChangeNewP(v) \wedge \neg ChangeNewD(v) \wedge CopyVar(v)$ **Then**
$p_v := newp_v; d_v := newd_v; Out_v := \emptyset$;
**If** $newd_v = 0$ **Then** $p_v := \emptyset; newp_v := \emptyset$;

---

### 4.4 Recovering phase

This subsection is dedicated to the description of the recovering phase. Recall that, since the system can start from an arbitrary configuration $\gamma$, edges which do not belong to any `MST` can be part a fragment in $\gamma$. Given a fragment $F$, the addition of an edge $e = (u, v)$ which do not belong to $F$ creates a unique cycle, called *fundamental cycle* related to $e$ and denoted $C_e$ (i.e., $C_e = \texttt{path}(u, nca(\ell_u, \ell_v)) \cup \texttt{path}(nca(\ell_u, \ell_v), v) \cup \{e\}$). Thus, the "red rule" may not be satisfied for every constructed fragment, i.e., for some fundamental cycle defined by an internal edge of a fragment the maximum edge weight belong to the fragment. To identify these edges, we verify that for each internal edge $e$ there is no edge in the fundamental cycle $C_e$ with a higher weight than $w(e)$. To this end, in a fragment $F$ the label of the nodes are used to identify the edges $e = \{u, v\}$ which do not belong to $F$ such that $u$ and $v$ have a common ancestor.

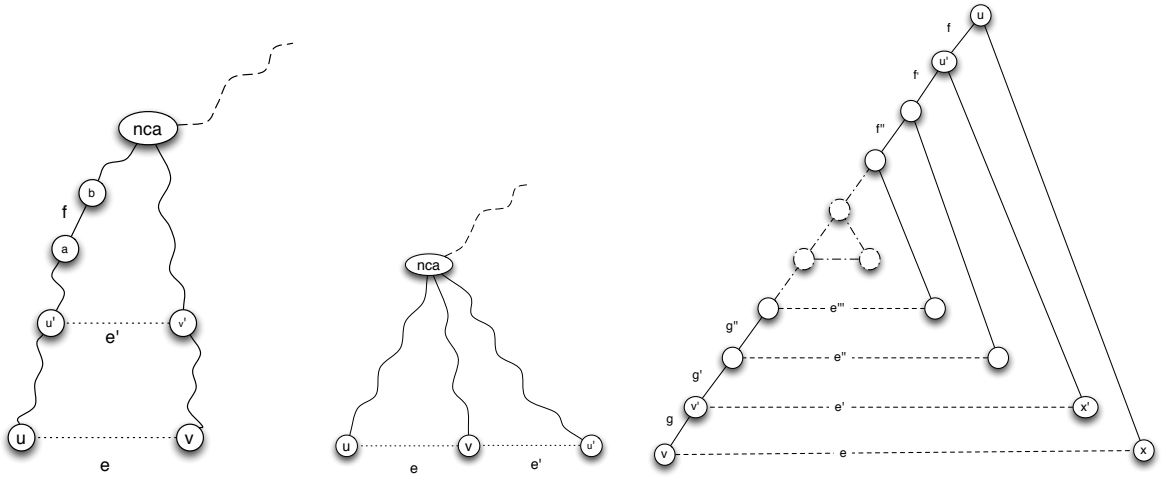


Figure 9: Use of internal edges for fundamental cycles verification

Let us consider a fragment $F$ and an edge $f = \{x, y\}$ belonging to $F$ such that $f \in C_e \setminus \{e\}$. If $w(f) > w(e)$ then $e$ must become an edge of $F$. Consequently, we need to verify all the edge weights of $C_e \setminus \{e\}$. To achieve this task, the weight of $e = (u, v)$ is sent up in $F$ along the two paths $\texttt{path}(u, nca(\ell_u, \ell_v))$ and $\texttt{path}(nca(\ell_u, \ell_v), v)$. Clearly, to maintain low space complexity, the nodes cannot store the information about all internal edges. Consequently, we decide that each node stores only the information of a single internal edge at a time. Specifically, we need to organize the circulation of the internal edges. A natural question to ask at this point is whether the information of all non-tree edges are needed. To answer to this question, we first make some observations.

First, suppose the following case (see Figure 9(a)): let $e = \{u, v\}$ and $e' = \{u', v'\}$ be internal edges such that $nca(\ell_u, \ell_v) = nca(\ell_{u'}, \ell_{v'})$, and $u'$ and $v'$ are closer to $nca(\ell_u, \ell_v)$ than $u$ and $v$. On $\texttt{path}(u', nca(\ell_u, \ell_v))$ and $\texttt{path}(v', nca(\ell_u, \ell_v))$ only the internal edge with the smallest weight is needed. To justify this assertion, let us consider without loss of generality that $w(e) < w(e')$ and $f = \{a, b\}$ is a tree edge such that $w(f) > w(e')$. Moreover, suppose that all edges in a $\texttt{path}(u, u')$ and $\texttt{path}(v, v')$ have a weight smaller than $w(e)$. Consequently, $f$ is not part of the MST, and if we delete $f$, the minimum outgoing edge of the fragment composed by the $\texttt{path}(a, u)$ is edge $e$. Consider now, the case when several adjacent edges of node $v$ have the same common ancestor (see Figure 9(b)). In this case only the internal edge with the smallest weight is relevant on the $\texttt{path}(v, nca(\ell_u, \ell_v))$ to avoid the maximum weight of the fundamental cycles. The last case considered is the following (see Figure 9(c)). Consider a path between two nodes $u$ and $v$, and $u', v' \in \texttt{path}(u, v)$ such that $f = \{u, u'\}$ and $g = \{v, v'\}$. Let $e = \{v, x\}$ be an edge such that $nca(\ell_v, \ell_x) = u$ and $e' = \{v', x'\}$ an edge such that $nca(\ell_{v'}, \ell_{x'}) = u'$. If $w(e') < w(e)$, the weight of $e'$ is needed to verify if the weights of the edges on $\texttt{path}(v', u')$ have a higher weight than $e'$. However, the weight of $e$ is needed to verify the weight of edge $f$. Consequently, we need to collect all the outgoing edges from the leaves to the root, from the farthest to the nearest of the root.

$$
\begin{array}{lcl}
IECA(v, ca) & = & \min\{w(u,v) : u \in N_v \setminus (\mathrm{C}(v) \cup \{p_v\}) \wedge nca(\ell_u, \ell_v) = ca \wedge nca(\ell_u, \ell_v) \succeq \ell_v\}^a \\
\mathrm{IE}_l(v) & = & \{(w(u,v), \ell_u, \ell_v) : u \in N_v \setminus (\mathrm{C}(v) \cup \{p_v\}) \wedge nca(\ell_u, \ell_v) \neq \emptyset \\
 & & \wedge w(u,v) = IECA(v, nca(\ell_u, \ell_v))\} \\
\mathrm{IE}_c(v) & = & \{In_u : u \in \mathrm{C}(v) \wedge nca(In_u[1], In_u[2]) \succeq \ell_v\} \\
minIE(v, ca) & = & \min\{e : e \in (\mathrm{IE}_l(v) \cup \mathrm{IE}_c(v)) \wedge nca(e[1], e[2]) \succ ca\} \\
\mathrm{IE}(v) & = & \begin{cases} minIE(v, nca(In_v[1], In_v[2])) & \text{If } minIE(v, nca(In_v[1], In_v[2])) \neq \emptyset \\ minIE(v, \perp) & \text{Otherwise} \end{cases} \\
EndForward(v) & \equiv & (p_v = \emptyset \vee nca(In_v[1], In_v[2]) = \ell_v) \wedge In_v = \mathrm{IE}_l(v) \\
Forwarded(v) & \equiv & (nca(In_v[1], In_v[2]) = \ell_v \vee In_{p_v} = In_v) \wedge In_v \notin \mathrm{IE}_c(v) \\
BetterEdgeP(v) & \equiv & p_v \neq \emptyset \wedge nca(In_v[1], In_v[2]) \neq \ell_v \\
 & & \wedge nca(In_{p_v}[1], In_{p_v}[2]) = nca(In_v[1], In_v[2]) \wedge In_{p_v} < In_v \\
SelectEdge(v) & \equiv & EndForward(v) \vee Forwarded(v) \vee BetterEdgeP(v) \\
Recover(v) & \equiv & p_v = newp_v \wedge d_v = newd_v \wedge newd_v = newd_{p_v} + 1 \wedge \mathrm{Candidate}(v) = \emptyset \\
 & & \wedge SelectEdge(v)
\end{array}
$$

[a] Operator $\succ$ is the lexicographical order used for the node labels, we consider $\perp$ as the smallest element.

Figure 10: Macros used by Algorithm SS-MST for the correction of the MST.

The rule for collecting the relevant internal edges is based on the above observations (see rule $\mathrm{R}_{\texttt{Rec}}$). The internal edges are sent up in the fragment from the leaves to the root using variable $In_v$ at every node $v$. The internal edges are collected locally by beginning from the edge with the farthest nearest common ancestor to the edge with the nearest common ancestor, i.e., following the lexicographical order on the nearest common ancestor labels and beginning by the smallest one. In case there exist several edges with the same nearest common ancestor, only the edge with the smallest weight is kept. The list of the ordered internal edges at node $v$ is given by Macro $\mathrm{IE}(v)$. This list is computed by different predicates (see Macros in Figure 10). Each node $v$ compares the weight stored in variable $In_v[0]$ with the weight of the edge leading to its parent. If $In_v[0] < w(v, p_v)$ then $v$ knows that the internal edge indicated by $In_v$ must belong to the MST. Consequently $v$ deletes the edge $(v, p_v)$ from the fragment (only if $v$ is not the nearest common ancestor of the internal edge given by $In_v$), and $v$ becomes the root of the new

fragment (see Rule $R_{\texttt{Rec}}$). A node $v$ can select a new internal edge by executing Rule $R_{\texttt{Rec}}$ in the following case (i.e., Predicate $Recover(v)$ is satisfied): (i) the internal edge of $v$ is propagated up by its parent and $v$ has no more child propagating the same internal edge (see Predicate $Forwarded(v)$), (ii) $v$ is the nearest common ancestor of the adjacent internal edge actually selected (see Predicate $EndForward(v)$), or (iii) $v$ is neither the root of the fragment nor the nearest common ancestor of the selected internal edge $e'$ and its parent propagates an internal $e''$ related with the same common ancestor but $w(e'') < w(e')$ (see Predicate $BetterEdgeP(v)$). This allows to obtain a piplined propagation of the internal edges. Figure 11 illustrates the bottom-up spread of the internal edges.

---

$R_{\texttt{Rec}}$**:** [ **Recovering** ]

**If** $CorrectF(v) \wedge Recover(v)$ **Then**

$In_v := \mathrm{IE}(v)$;

**If** $nca(In_v[1], In_v[2]) \neq \ell_v \wedge w(v, p_v) > In_v[0]$ **Then** $p_v := \emptyset; d_v := 0; \ell_v := (\mathsf{Id}_v, 0)$;

---

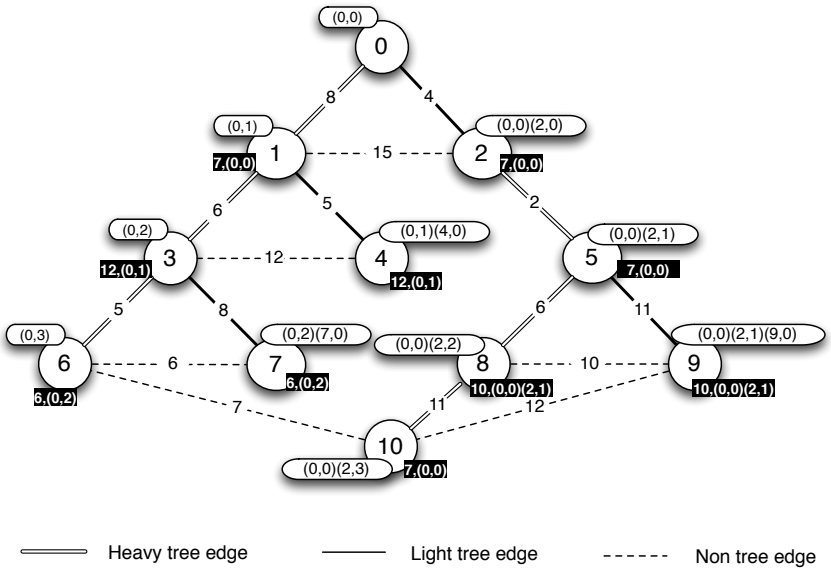


Figure 11: The white bubble at each node $v$ corresponds to the label of the node. The black bubble at each node represents the internal edges.

## 4.5 Correctness and complexity

This subsection is dedicated to the correctness of the self-stabilizing Minimum Spanning Tree construction. We can define a Minimum Spanning Tree as in Definition 3.

**Definition 3 (MST)** *Let $G = (V, E, w)$ be a network with $V$ the set of nodes, $E$ the set of undirected links and the function $w : E \to \mathbb{N}$. A graph $T = (V_T, E_T)$ of $G$ is called a* Minimum Spanning Tree *if the following conditions are satisfied:*

1. *$V_T = V$ and $E_T \subseteq E$, and*

2. *$T$ is a connected graph (i.e., there exists a path in $T$ between any pair of nodes $x, y \in V_T$) and $|E_T| = |V| - 1$, and*

3. *There exists no spanning tree $T'$ of $G$ whose the weight $w(T')$ is lower than $w(T)$.*

We give a formal specification to the problem of constructing a Minimum Spanning Tree, stated in Specification 1.

**Specification 1 (MST Construction)** *Let $\Gamma$ be the set of all possible configurations of the system. An algorithm $\mathcal{A}_{\mathcal{MST}}$ solving the problem of constructing a stabilizing MST tree satisfies the following conditions:*

[TC1] *Starting from any configuration in $\Gamma$, Algorithm $\mathcal{A}_{\mathcal{MST}}$ reaches in finite time a set of configurations $\mathcal{L} \subseteq \Gamma$ which satisfies Definition 3, and*

[TC2] *From every configuration $\gamma \in \mathcal{L}$, Algorithm $\mathcal{A}_{\mathcal{MST}}$ can only reach a configuration in $\mathcal{L}$.*

Let $\Gamma$ be the set of all possible configurations of the system. A fragment $\mathtt{F}$ rooted at node $r_{\mathtt{F}}$ is a subtree such that for every node $v \in \mathtt{F}$ there is a path to $r_{\mathtt{F}}$ and Predicate Distance($v$) is true. In the following theorem we start by showing that until a legitimate configuration is reached there is no deadlock in the system.

**Theorem 2** *Let the set of configurations $\mathcal{B} \subseteq \Gamma$ such that every configuration $\gamma \in \mathcal{B}$ satisfies Definition 3. $\forall \gamma \in (\Gamma - \mathcal{B}, \exists v \in V$ such that $v$ is enabled in $\gamma$.*

**Proof.** Assume by the contradiction, that $\exists \gamma \in (\Gamma - \mathcal{B})$ such that $\forall v \in V$ no rule is enabled at $v$ in $\gamma$. Since $\gamma \notin \mathcal{B}$, there is either a cycle, several fragments, or a single fragment which is not a MST in $\gamma$. If there is a cycle or incorrect distances in $\gamma$ then there exists a node $v$ such that $d_v \neq d_{p_v} + 1$. This implies that Predicate Distance($v$) is not satisfied and Rule $\mathrm{R}_{\mathtt{Correct}}$ is enabled at $v$, a contradiction. Otherwise, $\forall v \in V$ Predicate $Distance(v)$ is satisfied. If there exists a node $v$ in $\gamma$ with an incorrect label, then either Predicate SizeC($v$) or Label($v$) is satisfied and Rule $\mathrm{R}_{\mathtt{Size}}$ or $\mathrm{R}_{\mathtt{Label}}$ is enabled at $v$ (see proofs of Section 3.4 for more details), a contradiction. Otherwise, Predicate $CorrectF(v)$ is satisfied $\forall v$ in $\gamma$. If there are several fragments in $\gamma$ then there is at least one node $v \in V$ such that Macro $C$Candidate($v$) $\neq \emptyset$. If there is a node $v$ in $\gamma$ which has not computed the correct outgoing edge of its subtree (i.e., $Out_v[0] \neq$ Candidate($v$)), then Rule $\mathrm{R}_{\mathtt{Min}}$ is enabled at $v$, a contradiction. Otherwise, in each fragment $F$ in $\gamma$ we have $\forall v \in F, Out_v[0] =$ Candidate($v$). Consider first a node $v$ in a fragment $F$ in $\gamma$ which is on the path between the root of $F$ and the minimum outgoing edge of $F$ (i.e., $Out_{p_v} = Out_v$). If there exists such a node $v$ with $newp_v \neq NewParent(v)$ and Predicate $Reorientation(v)$ is satisfied, then Predicate $ChangeNewP(v)$ Rule $\mathrm{R}_{\mathtt{Merge}}$ is enabled at $v$, a contradiction. Otherwise, consider the other node $v$ in $F$ which are not on the path between the root and the minimum outgoing edge of $F$ (i.e., $Out_{p_v} \neq Out_v$). If there exist such a node $v$ such that $newp_v \neq p_v$ then Predicate $ChangeNewP(v)$ is satisfied and Rule $\mathrm{R}_{\mathtt{Merge}}$ is enabled at $v$, a contradiction. Otherwise in each fragment $F$ in $\gamma$, we have $\forall v \in V, (newp_v = NewParent(v) \vee newp_v = p_v) \Rightarrow \neg ChangeNewP(v)$. Either for the future root $v$ (i.e., $newp_{newp_v} = v$) of a fragment $F$ in $\gamma$ we have $newd_v > 1$ then Predicate $ChangeNewD(v)$ is satisfied and Rule $\mathrm{R}_{\mathtt{Dist}}$ is enabled at $v$, a contradiction. Or for the other node $v$ in $F$ we have $newd_v \neq newd_{p_v} + 1$ then Predicate $ChangeNewD(v)$ is satisfied and Rule $\mathrm{R}_{\mathtt{Dist}}$ is enabled at $v$, a contradiction. Otherwise, we have $\forall v \in V, \neg ChangeNewP(v) \wedge$

$ChangeNewD(v)$ in $\gamma$. If in a fragment $F$ in $\gamma$ there is a node $v$ such that every of its future children $u$ after the merging (given by Macro $NewChild(v)$) in the fragment satisfies $\neg CopyVar(u)$ and $p_v \neq newp_v \vee d_v \neq newd_v$, then Predicate $CopyVar(v)$ is satisfied and Rule $\mathrm{R}_{\texttt{End}}$ is enabled at $v$, a contradiction. Finally, otherwise there is only a single fragment $F$ in $\gamma$ and we have $\forall v \in F, CorrectF(v)$. Moreover, for every node $v \in V$ Predicate $Recover(v)$ is satisfied, since $(\neg ChangeNewP(v) \wedge \neg ChangeNewD(v) \wedge \neg CopyVar(v) \wedge CorrectF(v)) \Rightarrow Recover(v)$. Therefore, Rule $\mathrm{R}_{\texttt{Rec}}$ is enabled at every $v$ in $\gamma$. By contradiction the fragment $F$ in $\gamma$ is not a MST, so there exists a node $v$ in $\gamma$ such that $p_v \neq \emptyset$ and $v$ is adjacent of an internal edge with a weight lower than $w(v, p_v)$ (i.e.,$w(v, p_v) > In_v[0]$). Thus, $v$ becomes the root of a new fragment when $v$ executes Rule $\mathrm{R}_{\texttt{Rec}}$, a contradiction. □

We denote by $\Gamma_{\texttt{CF}}$ the set of configurations in $\Gamma$ such that there are no cycles in the subgraph induced by parent link relations (i.e., for every $\gamma \in \Gamma_{\texttt{CF}}$ we have $\forall v \in V, \text{Distance}(v)$).

**Lemma 5** *Starting from any arbitrary configuration, the system reaches in $O(n)$ rounds a configuration in $\Gamma_{\texttt{CF}}$.*

This lemma can be proved using the same arguments given in [8].

### 4.5.1 Correctness and complexity of the merging phase

We now define some notations and predicates which will be used in the following proofs. Given a configuration $\gamma \in \Gamma$, we note the set of all fragments in $\gamma$ by $\mathcal{F}(\gamma)$. Moreover, we define below several sets of fragments with different properties and the notion of *attractor*, introduced by Gouda and Multari [25], will be used to show that during the convergence of Algorithm `SS-MST` each fragment gains additional properties. We define five sets of fragments in a configuration $\gamma \in \Gamma$:

- Let $\mathcal{F}_1(\gamma) = \{F \in \mathcal{F}(\gamma) : (\forall v \in F : CorrectF(v))\}$ be the set of fragments in $\gamma$ in which all the nodes are correctly labeled.
- Let $\mathcal{F}_2(\gamma) = \{F \in \mathcal{F}_1(\gamma) : (\forall v \in F : Out_v[0] = \text{Candidate}(v) \neq \emptyset)\}$ be the set of fragments correctly labeled in $\gamma$ in which every node has computed its minimum-weight outgoing edge of its subtree for the merging phase.
- Let $\mathcal{F}_3(\gamma) = \{F \in \mathcal{F}_2(\gamma) : (\forall v \in F : \neg ChangeNewP(v))\}$ be the set of fragments correctly labeled in $\gamma$ in which every node has computed its future parent used when the merging phase is done.
- Let $\mathcal{F}_4(\gamma) = \{F \in \mathcal{F}_3(\gamma) : (\forall v \in F : \neg ChangeNewD(v))\}$ be the set of fragments correctly labeled in $\gamma$ in which every node has computed its future distance used when the merging phase is done.
- Let $\mathcal{F}_5(\gamma) = \{F \in \mathcal{F}_4(\gamma) : (\forall v \in F : \neg CopyVar(v))\}$ be the set of fragments in $\gamma$ for which the merging phase is done.

We obtain the following lemma by applying Rules $\mathrm{R}_{\texttt{Size}}$ and $\mathrm{R}_{\texttt{Label}}$ according to Lemmas 1 to 4 and Theorem 1.

**Lemma 6** *Starting from any configuration $\gamma \in \Gamma_{\texttt{CF}}$, after $O(n)$ rounds the system reaches a configuration $\gamma'$ such that for each fragment $F \in \mathcal{F}_\gamma$ we have $F \in F_1(\gamma')$.*

**Lemma 7** *Let any fragment $F \in \mathcal{F}_1(\gamma)$ in a configuration $\gamma \in \Gamma_{\texttt{CF}}$. In $O(h_{\mathcal{F}})$ rounds, we have $F \in \mathcal{F}_2(\gamma')$, with $\gamma' \in \Gamma_{\texttt{CF}}$ and $h_F$ the height of $F$.*

**Proof.** In the following we define the potential function $\mathcal{M}$. First, let $w_m$: $V \to \mathbb{N}$ be the function defined by:

$$w_m(v) = |\mathit{Out}_v[0] - \min\Big(\min\{\mathit{Out}_u[0] : u \in \mathrm{C}(v)\}, \min\{w\{v,u\} : (u,v) \in \mathrm{OE}(v)\}\Big)|.$$

Note that, we have $w_m \geq 0$. Variable $\mathit{Out}_v$ has a correct value at node $v$ if and only if $w_m = 0$. Let a fragment $F \in \mathcal{F}_1(\gamma)$ in a configuration $\gamma \in \Gamma_{\texttt{CF}}$, and $\mathcal{M} : \Gamma \to \mathbb{N}$ be the function defined by:

$$\mathcal{M}(\gamma) = \sum_{d=0}^{h_{\texttt{f}}} m_d(\gamma)(n+1)^d$$

where $m_d(\gamma)$ is the number of nodes $v$ at height $d$ in $F$ with $w_m(v) \neq 0$. We denote $n_F$ the number of nodes in fragment $F$. Note that $0 \leq m_d(\gamma) \leq n_F$, and $0 \leq \mathcal{M}(\gamma) \leq (n_F+1)^{h_F+1}$. Moreover, the variable $\mathit{Out}$ has a correct value at every node in $F$ if and only if $\mathcal{M}(\gamma) = 0$. We note $\gamma(t)$ the configuration of the system after round $t$. Let $d_0$ be the largest index such that $m_{d_0}(\gamma(t)) \neq 0$. Since we use a weakly fair scheduler, all the nodes of Fragment $F$ are scheduled during the execution of round $t+1$. Every node $v$ at height $d > d_0$ does not change the value of its variable $\mathit{Out}$ (see Rule $\mathrm{R}_{\texttt{Min}}$), and therefore $w_m(v)$ remains equal to zero, so $m_d(\gamma(t+1))$ is equal to zero as well. The nodes $v$ at height $d_0$ change their variable $\mathit{Out}_v$ according to the variable $\mathit{Out}_u$ of their children $u \in F$ (see Rule $\mathrm{R}_{\texttt{Min}}$). Let $v$ be a node at height $d_0$. The children of $v \in F$ (if any) are at height $d > d_0$. Thus, their variable $\mathit{Out}$ has not changed, and therefore $w_m(v)$ becomes zero after round $t+1$. As a consequence, $m_{d_0}(\gamma(t+1)) = 0$. Therefore, we get

$$\mathcal{M}(\gamma(t+1)) < \mathcal{M}(\gamma(t))$$

and thus the system will eventually reach a configuration where all the variables $\mathit{Out}$ contains the minimum outgoing edge of the sub-fragment rooted at $v \in F$ (see Predicate Candidate($v$)).

To measure the number of rounds it takes to converge, observe that $d$ decreases by at least one at each round. Since $d \leq h_F$, we get that starting from any configuration $\gamma \in \Gamma_{\texttt{CF}}$ with $F \in \mathcal{F}_1(\gamma)$ the system reaches a configuration where for every node $v$ in Fragment $F$ the variable $\mathit{Out}_v$ is correct after $O(h_F)$ rounds. □

**Lemma 8** *In every configuration $\gamma \in \Gamma_{\texttt{CF}}$, there are at least two fragments $F_1$ and $F_2$, $F_1, F_2 \in \mathcal{F}_2(\gamma)$ which select the same minimum-weight outgoing edge for merging.*

**Proof.** Assume, by the contradiction, that there exists a configuration $\gamma \in \Gamma_{\texttt{CF}}$ with less than two fragments in $\mathcal{F}_2(\gamma)$ which select the same minimum-weight outgoing edge. This implies in $\gamma$ that either at least one fragment $F \in \mathcal{F}(\gamma)$ which has not computed its minimum-weight outgoing edge, or every fragment $F \in \mathcal{F}_2(\gamma)$ has selected a different minimum-weight outgoing edge. In the former case, there is a contradiction since according to Lemma 7 in $O(h_F)$ additional rounds the system reaches a configuration $\gamma'$ in which at least two fragments $F_1$ and $F_2$, $F_1, F_2 \in \mathcal{F}_2(\gamma)$ which select the same minimum-weight outgoing edge for merging. Otherwise, let $|\mathcal{F}_2(\gamma)|$ denotes the number of fragments in the set $\mathcal{F}_2(\gamma)$ in $\gamma$. In the latter case, exactly $|\mathcal{F}_2(\gamma)|$ minimum-weight outgoing edges have been selected in $\gamma$. However, we can observe that we can define a total order on the outgoing edges in each configuration in $\Gamma_{\texttt{CF}}$ based on the tuple defined by the edges weight and the identifiers of the extremities of the edges. By using this total order, $n$ fragments could select at most the $n-1$ minimum outgoing edges.

Thus, since Algorithm SS-MST uses these method to select the minimum-weight outgoing edge of each fragment (see Macros Candidate($v$) and $NCand$(Candidate($v$))) then at most $|\mathcal{F}_2(\gamma)|-1$ different outgoing edges are selected in $\gamma$. So, there are at least two fragments $F_1, F_2 \in \mathcal{F}_2(\gamma)$ which select the same minimum outgoing edge, a contradiction. □

**Lemma 9** *Let any fragment $F, F \notin \mathcal{F}_3(\gamma)$, of a configuration $\gamma \in \Gamma_{\texttt{CF}}$. If $F \in \mathcal{F}_2(\gamma)$ then every computation suffix starting from $\gamma$ contains a configuration $\gamma' \in \Gamma_{\texttt{CF}}$ such that $F \in \mathcal{F}_3(\gamma')$.*

**Proof.** Assume, by the contradiction, that there exists a suffix $e''$ starting from $\gamma$ with no configuration $\gamma' \in \Gamma_{\texttt{CF}}$ such that $F \in \mathcal{F}_3(\gamma')$ in computation $e = e'e''$. Consider the configuration $\gamma$. Since $F \in \mathcal{F}_2(\gamma)$ and $F \notin \mathcal{F}_3(\gamma)$, only Rule $\mathrm{R}_{\texttt{Merge}}$ could be enabled at a node $v \in F$ (by definition of $\mathcal{F}_2(\gamma)$ and according to the guards of rules given in the formal description of Algorithm SS-MST). Moreover, as $F \notin \mathcal{F}_3(\gamma)$ there exists at least one node $v \in F$ such that Predicate $ChangeNewP(v)$ is satisfied at $v$. Consider a computation step $\gamma \mapsto \gamma''$ of $e$. Assume that Rule $\mathrm{R}_{\texttt{Merge}}$ is enabled at $v$ in $\gamma$ and not in $\gamma''$ but $v$ did not execute Rule $\mathrm{R}_{\texttt{Merge}}$. If $v$ is the root of $F$ (i.e., $p_v = \emptyset$) or $v$ is on the path between the root of $F$ and the selected minimum-weight outgoing edge then $\neg ChangeNewP(v)$ implies that $newp_v = NewParent(v)$, a contradiction since Rule $\mathrm{R}_{\texttt{Merge}}$ is the only rule in $\gamma$ which can change variable $newp_v$. Otherwise, for every other node $v \in F$, $\neg ChangeNewP(v)$ implies that $newp_{p_v} = p_v$, a contradiction since Rule $\mathrm{R}_{\texttt{Merge}}$ is the only rule in $\gamma$ which can change variable $newp_v$. By weakly-fairness assumption on the daemon, every node $v \in F$ executes Rule $\mathrm{R}_{\texttt{Merge}}$ and satisfies $\neg ChangeNewP(v)$.

Finally, we can observe that the set of fragments $\mathcal{F}_3(\gamma)$ is included in the set $\mathcal{F}_2(\gamma)$ by definition in a configuration $\gamma$. □

**Lemma 10** *Let any fragment $F \in \mathcal{F}_2(\gamma)$ in a configuration $\gamma \in \Gamma_{\texttt{CF}}$. In $O(h_F)$ rounds, we have $F \in \mathcal{F}_3(\gamma')$, with $\gamma' \in \Gamma_{\texttt{CF}}$ and $h_F$ the height of $F$.*

**Proof.** Let $d_F(v)$ denotes the height of $v$ in $F$. We show by induction the following proposition: In at most $O(j+1)$ rounds, we have $\forall v \in F, d_F(v) \leq j \Rightarrow ([(p_v = \emptyset \vee Out_{p_v} = Out_v) \Rightarrow newp_v = NewParent(v)] \vee newp_v = p_v)$.

In base case $j = 0$. Consider the root $v$ of Fragment $F$ (i.e., $p_v = \emptyset$). If $newp_v \neq NewParent(v)$ then Rule $\mathrm{R}_{\texttt{Merge}}$ is enabled at $v$ in round 0, since $(p_v = \emptyset \wedge newp_v \neq NewParent(v)) \Rightarrow ChangeNewP(v)$. Therefore, since the daemon is weakly fair then in the first configuration of round 1, $v$ executes Rule $\mathrm{R}_{\texttt{Merge}}$ and we have $newp_v = NewParent(v)$ at $v$ which verifies the proposition.

Induction case: We assume that in round $j = h_F - 1$ we have $\forall u \in F, d_F(v) \leq j \Rightarrow ([(p_v = \emptyset \vee Out_{p_v} = Out_v) \Rightarrow newp_v = NewParent(v)] \vee newp_v = p_v)$. We have to show that in round $j+1$ we have $\forall v \in F, d_F(v) \leq j+1 \Rightarrow ([(p_v = \emptyset \vee Out_{p_v} = Out_v) \Rightarrow newp_v = NewParent(v)] \vee newp_v = p_v)$. Consider any node $v \in F$ of height $j+1$ in $F$. By induction hypothesis, we have either $newp_{p_v} = NewParent(p_v) = v$ or $Out_{p_v} \neq Out_v$. In the former case, if $newp_v \neq NewParent(v)$ and $Out_v = Out_{p_v}$ then Rule $\mathrm{R}_{\texttt{Merge}}$ is enabled at $v$ in round $j$ (because $(newp_{p_v} = NewParent(p_v) = v \wedge Out_v = Out_{p_v} \wedge newp_v \neq NewParent(v)) \Rightarrow ChangeNewP(v)$). In the latter case, if $newp_v \neq p_v$ and $Out_v \neq Out_{p_v}$ then Rule $\mathrm{R}_{\texttt{Merge}}$ is enabled at $v$ in round $j$ (because $(Out_v \neq Out_{p_v} \wedge newp_v \neq p_v) \Rightarrow ChangeNewP(v)$). Thus, since the daemon is weakly fair then in the first configuration of round $j+1$ $v$ executes Rule $\mathrm{R}_{\texttt{Merge}}$. So, we have $([(p_v = \emptyset \vee Out_{p_v} = Out_v) \Rightarrow newp_v = NewParent(v)] \vee newp_v = p_v)$ at $v$. Therefore, in at most $O(h_F)$ rounds we have $\forall v \in F, d_F(v) \leq h_F \Rightarrow ([(p_v = \emptyset \vee Out_{p_v} = Out_v) \Rightarrow newp_v = NewParent(v)] \vee [Out_{p_v} \neq Out_v \wedge newp_v = p_v])$ and this implies that $\forall v \in F, \neg ChangeNewP(v)$. □

**Lemma 11** *Let any two fragments $F_1$ and $F_2$, $F_1, F_2 \notin \mathcal{F}_4(\gamma)$, of a configuration $\gamma \in \Gamma_{\texttt{CF}}$. If $F_1, F_2 \in \mathcal{F}_3(\gamma)$ and the same minimum-weight outgoing edge is selected by the two fragments then every computation suffix starting from $\gamma$ contains a configuration $\gamma' \in \Gamma_{\texttt{CF}}$ such that $F_1, F_2 \in \mathcal{F}_4(\gamma')$.*

**Proof.** Assume, by the contradiction, that there exists a suffix $e''$ starting from $\gamma$ with no configuration $\gamma' \in \Gamma_{\texttt{CF}}$ such that $F_1, F_2 \in \mathcal{F}_4(\gamma')$ in computation $e = e'e''$. As Rule $\mathrm{R}_{\texttt{Dist}}$ is the only rule to modify variable $newd_v$ such that $\neg ChangeNewD(v)$ is satisfied when executed, this implies that there exists a node $v \in (F_1 \cup F_2)$ which never executes Rule $\mathrm{R}_{\texttt{Dist}}$ in the computation suffix $e''$. Consider the configuration $\gamma$. According to the formal description of Algorithm SS-MST, Rules $\mathrm{R}_{\texttt{Correct}}, \mathrm{R}_{\texttt{Size}}, \mathrm{R}_{\texttt{Label}}, \mathrm{R}_{\texttt{Min}}, \mathrm{R}_{\texttt{Merge}}$, and $\mathrm{R}_{\texttt{Rec}}$ are disabled for any node $v \in (F_1 \cup F_2)$ since $F_1, F_2 \in \mathcal{F}_3(\gamma)$. Moreover, Rule $\mathrm{R}_{\texttt{End}}$ is disabled at any node $v \in (F_1 \cup F_2)$ as we have $d_u \neq newd_u$ for every node $u \in NewChild(v)$, because $F_1, F_2 \notin \mathcal{F}_4(\gamma)$ and $newd_v = \infty$ by the execution of Rule $\mathrm{R}_{\texttt{Merge}}$. So, only Rule $\mathrm{R}_{\texttt{Dist}}$ could be enabled at every node $v \in (F_1 \cup F_2)$. This implies that Rule $\mathrm{R}_{\texttt{Dist}}$ is disabled for every node $v \in (F_1 \cup F_2)$ (i.e., we have $\neg ChangeNewD(v)$). Consider without loss of generality Fragment $F = F_1$. Note that $F \in \mathcal{F}_3(\gamma), F \notin \mathcal{F}_4(\gamma)$, and $\forall v \in F$ we have $newd_v = \infty$ due to the execution of Rule $\mathrm{R}_{\texttt{Merge}}$. If $v \in F$ is the future root of $F$ (after the merging phase) then either $newp_{newp_v} \neq v$, a contradiction because $F_1$ and $F_2$ have selected the same minimum-weight outgoing edge, or $newd_v \leq 1$, a contradiction since $newd_v \neq \infty$. If $v \in F$ is any other node in $F$ then we have $(newd_v = newd_{newp_v} + 1) \Rightarrow \neg ChangeNewD(v)$, a contradiction since $newd_v \neq \infty$. Therefore, the system reaches a configuration $\gamma' \in \Gamma_{\texttt{CF}}$ in which for every node $v \in F$ we have $\neg ChangeNewD(v)$, so $F \in \mathcal{F}_4(\gamma')$.

Finally, we can observe that the set of fragments $\mathcal{F}_4(\gamma)$ is included in the set $\mathcal{F}_3(\gamma)$ by definition in a configuration $\gamma$. □

**Lemma 12** *Let any fragment $F \in \mathcal{F}_3(\gamma)$ in a configuration $\gamma \in \Gamma_{\texttt{CF}}$. In $O(h_F)$ rounds, we have $F \in \mathcal{F}_4(\gamma')$, with $\gamma' \in \Gamma_{\texttt{CF}}$ and $h_F$ the height of $F$.*

**Proof.** We can show by induction on the height of $F$ that in $O(h_F)$ rounds every node $v \in F$ satisfies $\neg NewChangeD(v)$) using the same method as in proof of Lemma 10. □

**Lemma 13** *Let any fragment $F, F \notin \mathcal{F}_5(\gamma)$, of a configuration $\gamma \in \Gamma_{\texttt{CF}}$. If $F \in \mathcal{F}_4(\gamma)$ then every computation suffix starting from $\gamma$ contains a configuration $\gamma' \in \Gamma_{\texttt{CF}}$ such that $F \in \mathcal{F}_5(\gamma')$.*

**Proof.** Assume, by the contradiction, that there exists a suffix $e''$ starting from $\gamma$ with no configuration $\gamma' \in \Gamma_{\texttt{CF}}$ such that $F \in \mathcal{F}_3(\gamma')$ in computation $e = e'e''$. Consider the configuration $\gamma$. Since $F \in \mathcal{F}_4(\gamma)$, only Rule $\mathrm{R}_{\texttt{End}}$ could be enabled at a node $v \in F$ (by definition of $\mathcal{F}_4(\gamma)$ and according to the guards of rules given in the formal description of Algorithm SS-MST). Moreover, as $F \notin \mathcal{F}_5(\gamma)$ there exists at least one node $v \in F$ such that Predicate $CopyVar(v)$ is satisfied at $v$. Consider a computation step $\gamma \mapsto \gamma''$ of $e$. Assume that Rule $\mathrm{R}_{\texttt{End}}$ is enabled at $v$ in $\gamma$ and not in $\gamma''$ but $v$ did not execute Rule $\mathrm{R}_{\texttt{End}}$. If $v \in F$ is a leaf and $v$ is not adjacent to the minimum-weight outgoing edge of $F$ then $\neg CopyVar(v)$ implies that $p_v = newp_v \wedge d_v = newd_v$ (because $v$ has no neighbor $u \in Child(v)$), a contradiction since Rule $\mathrm{R}_{\texttt{End}}$ is the only rule which could copy the value of $newp_v$ (resp. $newd_v$) in $p_v$ (resp. $d_v$). Otherwise for every other node $v \in F$, $\neg CopyVar(v)$ implies that either $p_v = newp_v \wedge d_v = newd_v$ or $\exists u \in NewChild(v)$ such that $d_u \neq newd_u \vee p_u \neq newp_u$). In the former case, there is a contradiction since Rule $\mathrm{R}_{\texttt{End}}$ is the only rule which could copy the value of $newp_v$ (resp. $newd_v$) in $p_v$ (resp. $d_v$). In the latter

case, there is a neighbor $u \in NewChild(v)$ which modified its variable $p_u$ or $d_u$ by executing Rule $\mathrm{R}_{\texttt{Correct}}$ or $\mathrm{R}_{\texttt{Rec}}$, a contradiction since only Rule $\mathrm{R}_{\texttt{End}}$ could be enabled at a node $v \in F$ in $\gamma$. By weakly-fairness assumption on the daemon, every node $v \in F$ executes Rule $\mathrm{R}_{\texttt{End}}$ and satisfies $\neg CopyVar(v)$.

Finally, we can observe that the set of fragments $\mathcal{F}_5(\gamma)$ is included in the set $\mathcal{F}_4(\gamma)$ by definition in a configuration $\gamma$. □

**Lemma 14** *Let any fragment $F \in \mathcal{F}_4(\gamma)$ in a configuration $\gamma \in \Gamma_{\texttt{CF}}$. In $O(h_F)$ rounds, we have $F \in \mathcal{F}_5(\gamma')$, with $\gamma' \in \Gamma_{\texttt{CF}}$ and $h_F$ the height of $F$.*

**Proof.** We can show by induction on the height of $F$ that in $O(h_F)$ rounds every node $v \in F$ satisfies $\neg CopyVar(v)$) using the same method as in proof of Lemma 10. □

**Lemma 15** *Let a configuration $\gamma \in \Gamma_{\texttt{CF}}$ such that $|\mathcal{F}(\gamma)| > 1$. We have $|\mathcal{F}(\gamma')| < |\mathcal{F}(\gamma)|$ for any configuration $\gamma' \in \Gamma_{\texttt{CF}}$ obtained after a merging step in every computation suffix starting from $\gamma$.*

**Proof.** Assume, by the contradiction, that there exists a suffix $e''$ starting from $\gamma$ with a configuration $\gamma' \in \Gamma_{\texttt{CF}}$ obtained after a merging step for which $|\mathcal{F}(\gamma')| \geq |\mathcal{F}(\gamma)|$ in computation $e = e'e''$. This implies that in $e''$ either there are no two fragments $F_1, F_2 \in \mathcal{F}(\gamma)$ which can merge together using the same minimum-weight outgoing edge, or $F_1$ and $F_2$ does not belong to the same fragment after a merging step. First of all, by Lemmas 6 and 7 every fragment $F \in \mathcal{F}(\gamma)$ which does not satisfies $CorrectF(v)$ and $Out_v[0] = \text{Candidate}(v)$ executes Rules $\mathrm{R}_{\texttt{Size}}, \mathrm{R}_{\texttt{Label}}$ and $\mathrm{R}_{\texttt{Min}}$ to belong to $\mathcal{F}_2(\gamma)$. So, we consider that every fragment in $\mathcal{F}(\gamma)$ belongs to $\mathcal{F}_2(\gamma)$. In the first case, this is a contradiction with Lemma 8 which shows that in $\gamma$ there are at least two fragments $F_1$ and $F_2$, $F_1, F_2 \in \mathcal{F}_2(\gamma)$ which select the same minimum-weight outgoing edge for merging. In the latter case, by Lemma 9 every fragment $F \in \mathcal{F}_2(\gamma)$ computes its future parent after the merging step, so $F_1, F_2 \in \mathcal{F}_3(\gamma)$. Moreover, by Lemma 11 we have $F_1, F_2 \in \mathcal{F}_4(\gamma')$ in $e''$. So, by Lemma 13 every node $v \in (F_1 \cup F_2)$ can execute Rule $\mathrm{R}_{\texttt{End}}$ and then Rules $\mathrm{R}_{\texttt{Size}}$ and $\mathrm{R}_{\texttt{Label}}$ to form a new fragment in $\mathcal{F}(\gamma')$ composed of $F_1$ and $F_2$ in $e''$, a contradiction. □

Note that for each fragment $F \in \mathcal{F}_5(\gamma)$ in $\gamma \in \Gamma_{\texttt{CF}}$, we have that each node $v \in F$ satisfies Predicate Distance($v$) but does not satisfies Predicate $CorrectF(v)$ (because of the merging step the labels of each node $v$ is no more correct). So, every node $v \in F$ can execute again Rules $\mathrm{R}_{\texttt{Size}}$ and $\mathrm{R}_{\texttt{Label}}$.

**Lemma 16** *Let any fragment $F \in \mathcal{F}_5(\gamma)$ of a configuration $\gamma \in \Gamma_{\texttt{CF}}$ such that $\forall v \in F, CorrectF(v)$. If $F$ does not span all the nodes of the system, then Rule $\mathrm{R}_{\texttt{Min}}$ is enabled in at least one node $v \in F$ in $\gamma$.*

**Proof.** First of all, Predicate $CorrectF(v)$ is satisfied at every node $v \in F$ because $F \in \mathcal{F}_5$. Assume, by the contradiction, that $F$ does not spans all the nodes of the system in $\gamma$ and Rule $\mathrm{R}_{\texttt{Min}}$ is disabled at every node $v \in F$. This implies that there is a node $v \in F$ which adjacent to an edge $(u, v)$ such that $u \notin F$. So, we have $\text{Cand}_l(v) \neq \emptyset \Rightarrow \text{Candidate}(v) \neq \emptyset$. Moreover, since $F \in \mathcal{F}_5$, $v$ has executed Rule $\mathrm{R}_{\texttt{End}}$ and we have $Out_v = \emptyset$. Therefore, Rule $\mathrm{R}_{\texttt{Min}}$ is enabled at $v$, a contradiction. □

**Lemma 17** *Starting from any configuration $\gamma \in \Gamma$ which contains several fragments, the system reaches a configuration $\gamma' \in \Gamma_{\texttt{CF}}$ which contains a single fragment spanning all the nodes of the system in $O(n^2)$ rounds, with $n$ the network size.*

**Proof.** First of all, according to Lemmas 5 and 6 in $O(n)$ rounds the system reaches a configuration $\gamma_1 \in \Gamma_{\texttt{CF}}$ in which each fragment $F \in \mathcal{F}_1(\gamma_1)$. Moreover, according to Lemmas 7, 10, 12 and 14 by summing up the complexities each merging step is performed using at most $O(n)$ rounds (since $n$ is an upper bound for the height of any fragment). Finally, in a configuration we can not have more than $n$ fragments so to obtain a spanning tree we can perform at most $n-1$ merging phases (attained when only two fragments can be merged at each step). Moreover, by Lemma 15 after each merging step the number of fragments is decreased by at least one. Therefore, by the above elements we obtain that a spanning tree is constructed in $O(n^2)$ rounds starting from an arbitrary configuration. □

**Lemma 18** *In every configuration $\gamma \in \Gamma_{\texttt{CF}}$ which contains a single fragment $T \in \mathcal{F}_5(\gamma)$ spanning all the nodes of the system and correctly labeled, then for every node $v \in T$ no rule of Algorithm* SS-MST*, except Rule* $\mathrm{R}_{\texttt{Rec}}$*, is enabled at $v$.*

**Proof.** Assume, by the contradiction, that there is an enabled rule, except Rule $\mathrm{R}_{\texttt{Rec}}$, of Algorithm SS-MST in a node $v \in T$ in $\gamma$. For every node $v \in T$ Predicate Distance($v$) is satisfied in $\gamma \in \Gamma_{\texttt{CF}}$ (by definition of $\Gamma_{\texttt{CF}}$), so Rule $\mathrm{R}_{\texttt{Correct}}$ is disabled at $v$, a contradiction. Since $T$ is correctly labeled, we have $\neg CorrectF(v) \Rightarrow (\mathrm{SizeC}(v) \wedge \mathrm{Label}(v))$, so Rules $\mathrm{R}_{\texttt{Size}}$ and $\mathrm{R}_{\texttt{Label}}$ are disabled at $v$, a contradiction. Since $T$ spans all the nodes of the system, for every node $v \in T$ we have $\mathrm{Candidate}(v) = \emptyset$ and Rule $\mathrm{R}_{\texttt{Min}}$ is disabled at $v$, a contradiction. Finally, we have that $T \in \mathcal{F}_5(\gamma)$ and by definition of $\mathcal{F}_5(\gamma)$ Rules $\mathrm{R}_{\texttt{Merge}}, \mathrm{R}_{\texttt{Dist}}, \mathrm{R}_{\texttt{End}}$, and $\mathrm{R}_{\texttt{Rec}}$ are disabled at $v$, a contradiction. □

### 4.5.2 Correctness and complexity of the recovering phase

**Lemma 19** *Let a configuration $\gamma \in \Gamma_{\texttt{CF}}$ such that $|\mathcal{F}(\gamma)| = 1$ and $F$ be the fragment of $\mathcal{F}(\gamma)$. At least one node $v \in F$ can execute Rule* $\mathrm{R}_{\texttt{Rec}}$*.*

**Proof.** First of all, we consider that every node $v \in F$ the distance and the label are correct (i.e., $CorrectF(v)$ and $\neg CopyVar(v)$ are satisfied which implies we have $p_v = newp_v \wedge d_v = newd_v \wedge newd_v = new_{p_v} + 1$). Assume, by the contradiction, that Rule $\mathrm{R}_{\texttt{Rec}}$ is disabled for every node $v \in F$. This implies that for every node $v \in F$ we have either $\mathrm{Candidate}(v) \neq \emptyset$, or another internal edge can not be selected. In the first case, by hypothesis of the lemma there is only one fragment in $\mathcal{F}(\gamma)$, so for every node $v \in F$ we have $\mathrm{Candidate}(v) = \emptyset$, a contradiction. In the second case, this implies that an internal edge of minimum-weight associated to the common ancestor of the edge is not propagated up in $F$. If $v$ is the common ancestor of a locally internal edge (given by Macro $\mathrm{IE}_l(v)$), then we have $EndForward(v) \Rightarrow Recover(v)$ and Rule $\mathrm{R}_{\texttt{Rec}}$ is enabled at $v$, a contradiction. If $v$ has selected a local internal edge (i.e., $In_v \in \mathrm{IE}_l(v)$ and $In_v \notin \mathrm{IE}_c(v)$) and the internal edge has been propagated by $p_v$ ($In_v = In_{p_v}$), then this implies we have $Forwarded(v) \Rightarrow Recover(v)$ and Rule $\mathrm{R}_{\texttt{Rec}}$ is enabled at $v$, a contradiction. Otherwise, for an internal edge propagated by a child $u$ in $F$ either $v$ is the common ancestor, or the internal edge has been propagated by $p_v$ ($In_v = In_{p_v}$) and $u$ has selected another internal edge (i.e., $In_v \notin \mathrm{IE}_c(v)$). This implies we have $Forwarded(v) \Rightarrow Recover(v)$ and Rule $\mathrm{R}_{\texttt{Rec}}$ is enabled at $v$, a contradiction. □

**Corollary 1** *In any configuration $\gamma \in \Gamma_{\texttt{CF}}$ such that $|\mathcal{F}(\gamma)| = 1$, by executing Rule $\mathrm{R}_{\texttt{Rec}}$ every node $v \in F$ sends up in $F$ the internal edges selected by $v$ and its descendants ordered locally on the nearest common ancestors (given by Macro $\mathrm{IE}(v)$), with $F$ the fragment in $\mathcal{F}(\gamma)$.*

**Lemma 20** *Let a configuration $\gamma \in \Gamma_{\texttt{CF}}$ such that $|\mathcal{F}(\gamma)| = 1$ and $F$ be the fragment of $\mathcal{F}(\gamma)$. Every internal edge related to the nearest common ancestor $x$ is not propagated up in $F$.*

**Proof.** According to Corollary 1, $x$ propagates up in $F$ the internal edges selected by its descendants and itself. Assume, by the contradiction, that the parent $x$ of the nearest common ancestor $y$ related to an internal edge $e \in E$ propagates up in $F$ the internal edge $e = (u, v)$, described by its weight $w(u, v)$ and the labels of its extremities $\ell_u$ and $\ell_v$ stored in variable $In_y$ at $y$. This implies that when $x$ executes Rule $\mathrm{R}_{\texttt{Rec}}$ then Macro $\mathrm{IE}(x)$ returns edge $e$. Macro $\mathrm{IE}(x)$ returns an edge from the union set of edges given by Macros $\mathrm{IE}_l(x)$ and $\mathrm{IE}_c(x)$. We must consider only Macro $\mathrm{IE}_c(x)$ since $y$ is the common ancestor of $e$. However, according to the formal description of Algorithm `SS-MST` Macro $\mathrm{IE}_c(x)$ contains only internal edges $f = (a, b)$ whose the nearest common ancestor related to $f$ has a label higher or equal to $x$'s label following the lexicographical order (i.e., $nca(\ell_a, \ell_b) \succeq \ell_x$). Thus, we have $e \notin \mathrm{IE}(x)$, a contradiction. □

**Lemma 21** *Let a configuration $\gamma \in \Gamma_{\texttt{CF}}$ such that $|\mathcal{F}(\gamma)| = 1$ and $F$ be the fragment of $\mathcal{F}(\gamma)$. If a node $v \in F$ selects by executing Rule $\mathrm{R}_{\texttt{Rec}}$ an internal edge $e = (x, y)$ such that $w(x, y) < w(v, p_v)$ and $v$ is not the common ancestor related to $(x, y)$ then the edge $(v, p_v)$ is deleted by $v$ from $F$.*

**Proof.** Assume, by the contradiction, that $v \in F$ selects by executing Rule $\mathrm{R}_{\texttt{Rec}}$ an internal edge $e = (x, y)$ such that $w(x, y) < w(v, p_v)$ but the edge $(v, p_v)$ is not deleted from $F$. According to the formal description of Algorithm `SS-MST`, a node can delete an edge of a fragment by executing Rule $\mathrm{R}_{\texttt{Rec}}$. So, this implies that by executing Rule $\mathrm{R}_{\texttt{Rec}}$ the edge $(v, p_v)$ is not deleted from $F$ by $v \in F$. Consider the internal edge $e = (x, y)$ stored in $In_v$ by executing Rule $\mathrm{R}_{\texttt{Rec}}$ at $v$ such that $In_v[0] < w(v, p_v)$. By description of Rule $\mathrm{R}_{\texttt{Rec}}$, $v$ does not delete the edge $(v, p_v)$ only if $nca(In_v[1], In_v[2]) = \ell_v$), which is a contradiction with the hypothesis of the lemma. □

**Lemma 22** *Starting from any configuration $\gamma \in \Gamma_{\texttt{CF}}$ which contains a single spanning tree $T$, the recovering phase is performed in $O(n^2)$ rounds, $n$ the network size.*

**Proof.** First of all, every node $v \in T$ sends up in $T$ the internal edge of minimum weight associated to each common ancestor $ca$, given by Macro $\mathrm{IE}(v)$ based on Macro $minIE(v, ca)$. Moreover, by Lemma 20 every internal edge $e$ is not propagated by the ancestors of the common ancestor $e$ in $T$. By Lemma 1, every node $v \in T$ sends up in the tree the internal edges selected by $v$ and its descendants ordered locally on the nearest common ancestors, that is following the lexicographical order on the label of nearest common ancestors. Observe that every node $v \in T$ is the common ancestor of at most $h_T$ internal edges selected to be propagated up in $T$, with $h_T$ the height of $T$. Furthermore, each propagated internal edge reaches its related nearest common ancestor in $O(h_T)$ rounds. However, the propagation of the internal edges is pipelined in $T$, since a node $v \in T$ can execute Rule $\mathrm{R}_{\texttt{Rec}}$ when its parent propagates its internal edge or the nearest common ancestor is reached (see Predicate $SelectEdge(v)$). Thus, for every nearest common ancestor $v \in T$ the propagation of the internal edges related to $v$ is performed in $O(h_T)$ rounds. Finally, there are at most $n$ nearest common ancestors in the spanning tree $T$, so the propagation of all the internal edges of $T$ is performed in $O(n.h_T) \leq O(n^2)$ rounds. □

**Lemma 23** *Starting from every configuration $\gamma \in \Gamma_{\texttt{CF}}$ satisfying Definition 3, the system can only reach a configuration $\gamma' \in \Gamma_{\texttt{CF}}$ which satisfies Definition 3.*

**Proof.** By Lemma 18, in every configuration $\gamma \in \Gamma_{\texttt{CF}}$ every rule of Algorithm SS-MST, except Rule $R_{\texttt{Rec}}$, is disabled at $v \in V$. Consider any configuration $\gamma \in \Gamma_{\texttt{CF}}$ which satisfies Definition 3. This implies that there is only a single spanning tree $T$ in $\gamma$ and in every fundamental cycle defined by each internal edge of $T$ Lemma 21 can not be applied. Therefore, by executing Rule $R_{\texttt{Rec}}$ at any node $v \in T$ no new fragment is created and the constructed minimum spanning tree $T$ is preserved. □

**Theorem 3** *Algorithm* SS-MST *is a self-stabilizing algorithm for Specification 1 under a weakly fair daemon with a convergence time of $O(n^2)$ rounds and memory complexity of $O(\log^2 n)$ bits per node, with $n$ the network size.*

**Proof.** We have to show first that starting from any configuration the execution of Algorithm SS-MST verifies Property [TC1] and [TC2] of Specification 1.

First of all, by Theorem 2 while the system does not reach a configuration satisfying Definition 3, there is a rule enabled, except Rule $R_{\texttt{Rec}}$, at a node $v \in V$. According to Lemmas 15, 21, 17 and 22, from any configuration Algorithm SS-MST reaches a configuration $\gamma \in \Gamma$ satisfying Definition 3 in finite time, which verifies Property [TC1]. Moreover, according to Lemma 23 from a configuration $\gamma \in \Gamma$ satisfying Definition 3 Algorithm SS-MST can only reach a configuration in $\Gamma$ satisfying Definition 3, which verifies Property [TC2] of Specification 1.

We consider now the convergence time and memory complexity of Algorithm SS-MST. According to Lemmas 17 and 22, each part of the algorithm (merging and recovering part) have a convergence time of at most $O(n^2)$ rounds to construct a minimum spanning tree. Moreover, Algorithm SS-MST maintains height variables at every node $v \in V$, composed of six variables of size $\log(n)$ bits (variables $p_v, d_v, newp_v, newd_v, size_v$, and $Out_v$) and two variables of size $O(\log^2(n))$ bits used to stored labels of nodes (variables $\ell_v$ and $In_v$). According to [20], Variable $\ell_v$ necessitates $\Theta(\log^2 n)$ bits of memory at every node $v \in V$. Therefore, no more than $O(\log^2(n))$ bits per node are necessary. □

# 5 Conclusion

We extended the Gallager, Humblet and Spira (GHS) algorithm, [9], to self-stabilizing settings via a compact informative labeling scheme. Thus, the resulting solution presents several advantages appealing for large scale systems: it is compact since it uses only poly-logarithmic in the size of the network memory space ($O(\log^2(n))$ bits per node) and it scales well since it does not rely on any global parameter of the network. The convergence time of the proposed solution is $O(n^2)$ rounds. Quite recently, another self-stabilizing algorithm was proposed by Korman et al. [16] for the MST problem with a convergence time of $O(n)$ rounds and memory complexity of $O(\log(n))$ bits. However, this approach requires the use of several sub-algorithms leading to a complex solution to be used in a practical situation, comparing to our algorithm.

# References

[1] Stephen Alstrup, Cyril Gavoille, Haim Kaplan and Theis Rauhe . Nearest common ancestors: a survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, 2004.

[2] Doina Bein, Ajoy Kumar Datta and Vincent Villain. Self-Stablizing Pivot Interval Routing in General Networks. *ISPAN*, pages 282–287, 2005.

[3] Lélia Blin, Shlomi Dolev, Maria Gradinariu Potop-Butucaru and Stephane Rovedakis Fast Self-stabilizing Minimum Spanning Tree Construction - Using Compact Nearest Common Ancestor Labeling Scheme. *24th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 480–494, 2010.

[4] Lélia Blin, Maria Potop-Butucaru, Stephane Rovedakis and Sébastien Tixeuil. A New Self-stabilizing Minimum Spanning Tree Construction with Loop-Free Property. *23rd International Symposium on Distributed Computing (DISC)*, volume 5805 of *Lecture Notes in Computer Science*, pages 407–422. Springer 2009.

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

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

[7] Janna Burman and Shay Kutten. Time Optimal Asynchronous Self-stabilizing Spanning Tree. *21st International Symposium on Distributed Computing (DISC)*, volume 4731 of *Lecture Notes in Computer Science*, pages 92-107. Springer 2007.

[8] Shlomi Dolev, Amos Israeli and Shlomo Moran, Uniform Dynamic Self-Stabilizing Leader Election. IEEE Trans. Parallel Distrib. Syst., volume 8-4, pages 424–440, 1997.

[9] 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.

[10] Gheorghe Antonoiu and Pradip K. Srimani. Distributed Self-Stabilizing Algorithm for Minimum Spanning Tree Construction. *3rd International Conference on Parallel and Distributed Computing (Euro-Par)*, volume 1300 of *Lecture Notes in Computer Science*, pages 480-487. Springer 1997.

[11] 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.

[12] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal Computing*, 13(2):338-355, 1984.

[13] Lisa Higham and Zhiying Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *15th International Conference on Distributed Computing (DISC)*, volume 2180 of *Lecture Notes in Computer Science*, pages 194–208, 2001.

[14] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[15] Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. In *Distributed Computing*, 20(4): pages 253–266, 2007.

[16] Amos Korman and Shay Kutten and Toshimitsu Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an MST. In *30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 311–320, 2011.

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

[18] Jungho Park, Toshimitsu Masuzawa, Kenichi Hagihara and Nobuki Tokura. Distributed Algorithms for Reconstructing MST after Topology Change. *4th International Workshop on Distributed Algorithms (WDAG)*, pages 122–132, 1990.

[19] Jungho Park, Toshimitsu Masuzawa, Ken'ichi Hagihara and Nobuki Tokura. Efficient distributed algorithm to solve updating minimum spanning tree problem. *Systems and Computers in Japan*, 23(3):1–12, 1992.

[20] David Peleg, Informative Labeling Schemes for Graphs, *MFCS*, pages 579–588, 2000.

[21] David Peleg, *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, 2000.

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

[23] R. E. Tarjan, Data Structures and Network Algorithms. SIAM, volume 44, 1983.

[24] Gerard Tel. *Introduction to distributed algorithm.* Cambridge University Press, Second edition, 2000.

[25] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing Communication Protocols. IEEE Trans. Computers, volume 40(4), pages 448-458, 1991.

[26] Paola Flocchini and Toni Mesa Enriquez and Linda Pagli and Giuseppe Prencipe and Nicola Santoro. Distributed Computation of All Node Replacements of a Minimum Spanning Tree. *13th International Conference on Parallel and Distributed Computing (Euro-Par)*, volume 4641 of *Lecture Notes in Computer Science*, pages 598-607. Springer 2007.

[27] Paola Flocchini and Toni Mesa Enriquez and Linda Pagli and Giuseppe Prencipe and Nicola Santoro. Distributed Minimum Spanning Tree Maintenance for Transient Node Failures. *IEEE Trans. Computers*, 61(3):408–414, 2012.