Fast Self-Stabilizing Minimum Spanning Tree Construction

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

Fast Self-Stabilizing Minimum Spanning Tree Construction

Using Compact Nearest Common Ancestor Labeling Scheme

Lélia Blin[1,3] Shlomi Dolev[4] Maria Gradinariu Potop-Butucaru[2,3,5]
Stephane Rovedakis[1,6]

July 21, 2010

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 all 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.

1 Introduction

Since its introduction in a centralized context [15, 14], 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, often subject to transient faults, revives the study of scalable and self-stabilizing 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). *Self-stabilization* introduced first by Dijkstra in [6] and later publicized by several books [7, 8] 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 exists 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 logarithmic memory and preserves all the good characteristics of the original solution in terms of convergence time and scalability.

Gupta and Srimani presented in [13] 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

[1]Université d'Evry-Val d'Essonne, 91000 Evry, France.
[2]Université Pierre & Marie Curie - Paris 6, 75005 Paris, France.
[3]LIP6-CNRS UMR 7606, France. {lelia.blin,maria.gradinariu}@lip6.fr
[4]Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel. dolev@cs.bgu.ac.il
[5]INRIA REGAL, France.
[6]Laboratoire IBISC-EA 4526, 91000 Evry, France. stephane.rovedakis@ibisc.fr

of all paths from it to all the other nodes. To implement this algorithm, the authors assume that every node knows the number n of nodes in the network, and that the identifiers of the nodes are in $\{1, \ldots, n\}$. Every node u stores the weight of the edge $e_{u,v}$ placed in the MST for each node $v \neq u$. Therefore the algorithm requires $\Omega(\sum_{v \neq u} \log w(e_{u,v}))$ bits of memory at node u. Since all the weights are distinct integers, the memory requirement at each node is $\Omega(n \log n)$ bits. 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 also that the time complexity announced by the authors, $O(n)$ stays only in the particular synchronous settings considered by the authors. In asynchronous setting the complexity is $\Omega(n^2)$ rounds. A different approach for the message-passing model, was proposed by Higham and Liang [11]. The algorithm performs roughly as follows: every edge checks whether it eventually belongs 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 [13]. This solution also assume 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 diameter of the network respectively, i.e., $O(n^3)$ rounds in the worst case.

In [2] 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 unicity 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 [11].

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, where a boldface denotes the most useful (or efficient) feature for a particular criterium). 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. Therefore, we extend the GHS algorithm to self-stabilizing settings and keep compact its memory space by using a self-stabilizing extension of the nearest common ancestor labeling scheme of [1]. Note that labeling schemes have already been used in order to infer a broad set of information such as vertex adjacency, distance, tree ancestry or tree routing [5], however none of these schemes have been studied in self-stabilizing settings (except the last one).

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. Moreover, our solution is the best space/time compromise over the existing self-stabilizing MST solutions. 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.

	a priori knowledge	space complexity	convergence time
[13]	network size and the nodes in the network	$O(n \log n)$	$\Omega(n^2)$
[11]	upper bound on diameter	$O(\log n)$ +messages of size $O(n \log n)$	$O(n^3)$
[2]	**none**	$\mathbf{O(\log n)}$	$O(n^3)$
This paper	**none**	$O(\log^2 n)$	$\mathbf{O(n^2)}$

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

2 Model and notations

We consider an undirected weighted connected network $G = (V, E, w)$ where V is the set of nodes, E is the set of edges and $w : E \to \mathbb{R}^+$ is a positive cost function. Nodes represent processors and edges represent bidirectional communication links. Additionally, we consider that $G = (V, E, w)$ is a network in which the weight of the communication links may change value.

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

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

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

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

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

3 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 for large scale systems: it is compact since it uses only logarithmic memory in the size of the network, it scales well since it does not rely on any global parameter of the system, it is fast — its time complexity is the better known in self-stabilizing settings. Additionally, it self-recovers from any transient fault.

The central notion in the GHS approach is the notion of *fragment*. A fragment is a partial spanning 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 unique endpoint in F. The minimum-weight outgoing edge of a fragment F is 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 $\texttt{ME}_F$. Note that, with this scheme, more than two fragments may be merged concurrently. The merging process is recursively repeated until a single fragment remains. The result is a MST. The above approach is often called "blue rule" for MST construction.

This approach is particularly appealing when transient faults yield to 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 variables may be corrupted leading to a configuration of the network where the set of fragments are not sub-trees of some MST. That is, it may be a spanning tree but not of minimum weight, or it can contain cycles. In this case, the application of the blue rule only is not 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". The red rule removes the heaviest edge from every cycle. The resulting configuration contains a MST. We use the red rule as follows: given a spanning tree T of G, every edge e of G that is not in T is added to T, thus creating a (unique) cycle in $T \cup \{e\}$. This cycle is called a *fundamental cycle*, denoted by C_e. If e is not the edge of maximum weight in C_e, then, according to the red rule, there exists an edge $f \neq e$ in C_e with $w(f) > w(e)$. The edge of maximum weight can be removed since it is not part of any MST.

Our `MST` construction combines both the blue and red rules. The blue rule application needs that each node identifies its own fragment. The red rule requires that nodes identify the fundamental cycle corresponding to every adjacent non-tree-edge. In both cases, we use a self-stabilizing labeling scheme, called `NCA-L`, which provides at each node a distinct informative label such that the nearest common ancestor of two nodes can be identified based only on the labels of these nodes (see Section 3.1). Thus, the advantage of this labeling is twofold. First the labeling helps nodes to identify their fragments. 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, our algorithm will use the blue rule to construct 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 [3, 4] 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.

Variables used by `NCA-L` and `MST` modules For any node $v \in V(G)$, we denote by $N(v)$ the set of all neighbors of v in G. We use the following notations:

- p_v: *the parent* of v in the current spanning tree, an integer pointer to a neighbor;

- ℓ_v: *the label* of v composed of a list of pairs of integers where each pair is an identifier and a distance (the size of ℓ_v is bounded by $O(\log^2 n)$ bits);
- $size_v$: a pair of variables, the first one is an integer *the number of nodes in the sub-tree* rooted at v and the second one is the identifier of the child u of v with the maximum number of nodes in the sub-tree rooted at u;
- mwe_v: the *minimum weighted edge* composed by a pair of variables, the first one is an integer, the weight of the edge and the second one is the label of a node u stored in ℓ_u.

3.1 Self-stabilizing Nearest Common Ancestor Labeling

Our labeling scheme, called in the following `NCA-L`, uses the notions of *heavy* and *light* edges introduced in [10]. In a tree, a *heavy* edge is an edge between a node u and one of its children v of maximum 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, otherwise v is called *light node*. Moreover, the root of a tree is a heavy node. The idea of the scheme is as follows. A tree is recursively divided into disjoint paths: the heavy and the light paths which contain only heavy and light edges respectively.

- $\text{Child} \equiv \{u \in N(v) : p_u = \mathsf{Id}_v\}$
- $\text{SizeC}(v) \equiv \text{Leaf}(v) \vee (size_v = (1 + \sum_{u \in \text{Child}(v)} size_u, \arg\max\{size_u : u \in \text{Child}(v)\}))$
- $\text{Leaf}(v) \equiv (\nexists u \in N(v), p_u = \mathsf{Id}_v) \wedge size_v = (1, \perp)$
- $\text{Label}(v) \equiv \text{Label}_R(v) \vee \text{Label}_{Nd}(v)$
- $\text{Label}_R(v) \equiv (p_v = \emptyset \wedge \ell_v = (\mathsf{Id}_v, 0))$
- $\text{Label}_{Nd}(v) \equiv p_v \in N(v) \wedge (\text{Heavy}(v) \vee \text{Light}(v))$
- $\text{Heavy}(v) \equiv size_{p_v}[1] = \mathsf{Id}_v \wedge size_v[0] < size_{p_v}[0] \wedge \mathsf{last}(\ell_{p_v})[1] + 1 = \mathsf{last}(\ell_v)[1]$
- $\text{Light}(v) \equiv size_{p_v}[1] \neq \mathsf{Id}_v \wedge size_v[0] \leq size_{p_v}[0]/2 \wedge \ell_v = \ell_{p_v}.(\mathsf{Id}_v, 0)$
- $nca(u,v) \equiv \begin{cases} \ell.(a_0,a_1) & \text{s.t. } \ell_u \cap \ell_v = \ell, \ell_u = \ell.(a_0,a_1).\ell'_u \text{ and } \ell_v = \ell.(b_0,b_1).\ell'_v \quad \textbf{if } (a_0 = b_0 \vee \ell \neq \emptyset) \wedge \ell_u \prec \ell_v \\ \ell.(b_0,b_1) & \text{s.t. } \ell_u \cap \ell_v = \ell, \ell_u = \ell.(a_0,a_1).\ell'_u \text{ and } \ell_v = \ell.(b_0,b_1).\ell'_v \quad \textbf{if } (a_0 = b_0 \vee \ell \neq \emptyset) \wedge \ell_v \prec \ell_u \\ \emptyset & \textbf{otherwise} \end{cases}$
- $\text{Cycle}(v) \equiv \ell_v \subset \ell_{p_v} \vee \ell_v \prec \ell_{p_v}$
- $\text{MinEnabled}(v) \equiv \text{Enabled}(v) \wedge (\forall u \in N(v), \text{Enabled}(u) \wedge \mathsf{Id}_v < \mathsf{Id}_u)$

Figure 1: Predicates used by the algorithm `NCA-L` for the labeling procedure.

To label the nodes in a tree T, the size of each subtree rooted at each node of T is needed to identify heavy edges leading the heaviest subtrees at each level of 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. The second integer is the

identifier of a child of v with maximum number of nodes. That is, it indicates the heavy edge. The computation of $size_v$ is processed from the leaves to the root (see Predicate SizeC(v) in Figure 1). A leaf has no child, therefore $size_v = (1, \perp)$ for a leaf node v (see Predicate Leaf(v) in Figure 1 and Rule R_ℓ).

Based on the heavy and light nodes in a tree T indicated by variable $size_v$ at each node $v \in T$, each node of T can compute its label. The label of a node v stored in ℓ_v is a list of pair of integers. Each pair of the list contains the identifier of a node and a distance to the root of the heavy path (i.e., a path including only heavy edges). For the root v of a fragment, the label ℓ_v is the following pair $(\mathsf{Id}_v, 0)$, respectively the identifier of v and the distance to itself, i.e., zero (see Rule $R_\odot$). 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. 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. When 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 label of its parent to which it adds a pair of integers composed of its identifier and a zero distance (see Figure 2).

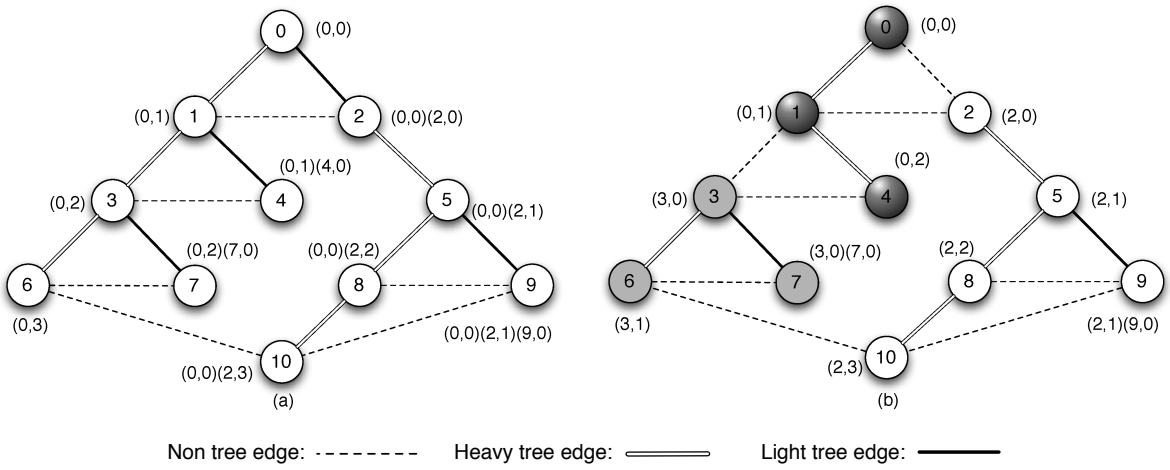

Figure 2: Labeling scheme

This labeling scheme is used in second part of this article in `MST` algorithm to find the minimum weighted edges, but it is also used to detect and destroy cycles since the initial configuration may not be a spanning tree. To this end, we define an order $\prec$ on the labels of nodes. Let a and b be two nodes and ℓ_a and ℓ_b be their respective labels such that $\ell_a = \ell.(a_0, a_1).\ell'_a$ and $\ell_b = \ell.(b_0, b_1).\ell'_b$ with $\ell_a \cap \ell_b = \ell$. The label of a node a is lower than the label of node b, noted $\ell_a \prec \ell_b$, if (1) $(a_0, a_1).\ell'_a = \emptyset$ and $(b_0, b_1).\ell'_b \neq \emptyset$, or (2) $a_0 < b_0$ or (3) $a_0 = b_0$ and $a_1 < b_1$.

A node u can detect the presence of a cycle by only comparing its label with the label of its parent. That is, if its label is contained in the label of its parent, or it is inferior to the one of its parent then u is part of a cycle (see Predicate Cycle(v) in Figure 1). In this case, the node u becomes the root of its fragment in order to break the cycle (see below Rule $R_\odot$).

Algorithm `NCA-L` is composed of two rules. Rule $R_\odot$ creates a root or breaks cycles while rule R_ℓ produces a proper labeling. Note that the last predicates in rules $R_\odot$ and R_ℓ (the part in gray) are used only for insuring the exclusivity of rules execution when the labeling scheme works together with the `MST` scheme.

A node v with an incoherent parent (which is not one of its neighbors) or present in a cycle executes $R_{\odot}$. Following the execution of this rule node v becomes a root node, it sets its parent to void and its label to $(\mathsf{Id}_v, 0)$.

Rule R_ℓ helps a node v to compute the number of nodes in its sub-tree (stored in variable $size_v$) and provides to v a coherent label.

$R_{\odot}$**:** [**Root creation**]
If $p_v \notin N(v) \vee (p_v = \emptyset \wedge \ell_v \neq (\mathsf{Id}_v, 0) \vee (\text{Cycle}(v) \wedge \neg\text{NeedReorientation}(v))$
Then $p_v := \emptyset;\ \ell_v = (\mathsf{Id}_v, 0);$

R_ℓ**:** [**Label correction**]
If $\neg\text{Cycle}(v) \wedge (\neg\text{SizeC}(v) \vee \neg\text{Label}(v)) \wedge \text{MinEnabled}(v) \wedge \neg\text{TreeMerg}(v)$
Then If Leaf(v) **then** $size_v = (1, \perp)$
Else $size_v := (1 + \sum_{u \in \text{Child}(v)} size_u, \arg\max\{size_u : u \in \text{Child}(v)\});$
If $size_{p_v}[1] = \mathsf{Id}_v$ **then** $\ell_v := \ell_{p_v};\ last(\ell_v)[1] := last(\ell_v)[1] + 1;$
Else $\ell_v = \ell_{p_v}.(\mathsf{Id}_v, 0)$

3.2 Self-stabilizing MST

In this section we describe our self-stabilizing MST algorithm. The algorithm executes two phases: the MST correction and the MST fragments merging. Recall that our algorithm uses the blue rule to construct a spanning tree and the red rule to recover from invalid configurations. In both cases, it uses the nearest-common ancestor labeling scheme to identify fragments and fundamental cycles. We assume in the following that the *merging* operations have a higher priority than the *recovering* operations. That is, the system recovers from an invalid configuration if and only if no merging operation is possible. In the worst case, after a failure hit the system, a merging phase will be followed by a recovering phase and finally by a final merging phase.

3.2.1 The minimum weighted edge and MST correction

Note that the scope of our labeling scheme is twofold. First, it allows a node to identify the neighbors that share the same fragment and consequently to select the outgoing edges of a fragment. Second, the labeling scheme may be used to identify cycles and to repair the tree. To this end, the algorithm uses the nearest common ancestor predicate *nca* depicted in Figure 1. For two nodes u and v with $e = (u, v)$ a non tree edge (i.e., $p_u \neq v$ and $p_v \neq u$), if the nearest common ancestor does not exist then u and v are in two distinct fragments (i.e., if we have $nca(u, v) = \emptyset$). Otherwise u and v are in the same fragment F and the addition of e to F generates a cycle. Let $\texttt{path}(x, y)$ be the set of edges on the unique path between x and y in F, with $x, y \in F$. The fundamental cycle C_e is the following: $C_e = \texttt{path}(u, nca(u, v)) \cup \texttt{path}(nca(u, v), v) \cup e$. Consider the example depicted on Figure 2(b). The labels of nodes 10 and 6 are respectively $\ell_{10} = (2, 3)$ and $\ell_6 = (3, 1)$. In this case $nca(10, 6) = \emptyset$ so the edge $(10, 6)$ is an outgoing edge because the nodes 10 and 6 are in two distinct fragments and they have no common ancestor. If the edge $(10, 6)$ is of minimum weight then $(10, 6)$ can be used for a merging between the fragment rooted in 2 and the fragment rooted in 3. For the case of nodes 10 and 9 the labels are $\ell_{10} = (2, 3)$ and $\ell_9 = (2, 1)(9, 0)$ and $nca(10, 9) = (2, 1)$. Consequently, 10 and 9 are in the same fragment. The fundamental cycle C_e with $e = (9, 10)$ goes through the node with the label $nca(10, 9)$, in other word the node 5 in Figure 2(b).

Predicate MinEdge(v) (see Figure 3) computes both the minimum weight outgoing edge used in a merging phase and the internal edges used in a recovering phase. Our algorithm

- $\text{MergeChild}(v) \equiv \min\{mwe_u : u \in \text{Child}(v) \wedge mwe_u[1] = \emptyset\}$
- $\text{MergeAdj}(v) \equiv (\min\{w(u,v) : u \in N(v) \setminus \text{Child}(v) \setminus \{p_v\} \wedge nca(u,v) = \emptyset\}, \emptyset)$
- $\text{MergeEdge}(v) \equiv \min\{\text{MergeChild}(v), \text{MergeAdj}(v)\}$
- $\text{FarLcaChild}(v) \equiv \arg\min_{\prec}\{mwe_u[1] : u \in \text{Child}(v) \wedge mwe_u[1] \preceq \ell_v\}$
- $\text{RecoverChild}(v) \equiv mwe_u$ such that $\text{FarLcaChild}(v) = u$
- $\text{FarLca}(v) \equiv \arg\min_{\prec}\{nca(u,v) : u \in N(v) \setminus \text{Child}(v) \setminus \{p_v\} \wedge mwe_v[1] \neq \emptyset \wedge nca(u,v) \succ mwe_v[1] \wedge nca(u,v) \neq \emptyset\}$
- $\text{RecoverAdj}(v) \equiv \begin{cases} (w(u,v), nca(u,v)) \text{ s.t. } \text{FarLca}(v) = u & \textbf{if } \text{FarLca}(v) \neq \emptyset \\ (w(u,v), nca(u,v)) & \textbf{otherwise} \\ \text{s.t. } u = \arg\min_{\prec}\{nca(u,v) : & \\ u \in N(v) \setminus Child(v) \setminus \{p_v\} \wedge nca(u,v) \neq \emptyset\} & \end{cases}$
- $\text{RecoverEdge}(v) \equiv \begin{cases} \text{RecoverChild}(v) & \textbf{if } \text{RecoverAdj}(v)[1] \prec \text{RecoverChild}(v) \\ \text{RecoverAdj}(v) & \textbf{otherwise} \end{cases}$
- $\text{MinEdge}(v) \equiv \begin{cases} \text{MergeEdge}(v) & \textbf{if } \text{MergeEdge}(v) \neq \emptyset \\ \text{RecoverEdge}(v) & \textbf{otherwise} \end{cases}$
- $\text{NeedReorientation}(v) \equiv \text{Label}_R(v) \vee (\ell_{p_v} = (\perp, \perp) \wedge p_{p_v} = \mathsf{Id}_v)$
- $\text{EndReorientation}(v) \equiv \ell_{p_v} = (\emptyset, \emptyset) \wedge (\ell_v = (\perp, \perp) \vee \ell_v \neq (\emptyset, \emptyset))$
- $\text{TreeMerg}(v) \equiv \text{NeedReorientation}(v) \vee \text{EndReorientation}(v)$
- $\text{NewFrag}(v) \equiv mwe_v = mwe_{p_v} \wedge mwe_v[1] \neq \mathsf{Id}_v \wedge w(v, p_v) > mwe_v[0]$

Figure 3: Predicates used by the MST for the tree correction or the fusion fragments.

gives priority to the computation of minimum outgoing edges via Predicate MinEdge(v). A recovering phase is initiated if there exists a unique tree or if a sub-tree of one fragment has no outgoing edge.

The computation of the minimum weight outgoing edge is done in a fragment F_u if and only an adjacent fragment F_v is detected by F_u, i.e., if we have Predicate MergeEdge(v) $\neq \emptyset$. In this case, using Rule R_{Min} each node collects from the leaves to the root the outgoing edges leading to an adjacent fragment F_v. At each level in a fragment, a node selects the outgoing edge of minimum weight among the outgoing edges selected by its children and its adjacent outgoing edges. Thus, this allows to the root of a fragment to select the minimum outgoing edge e of the fragment leading to an adjacent fragment. Then, the edge e can be used to perform a merging between two adjacent fragments using an edge belonging to a MST.

Let us explain Rule R_{ℓ} which allows to correct a tree (or a fragment). In this case, the information about the non-tree edges are sent to the root as follows. Among all its non-tree edges, a node u sends the edge $e = (u, v)$ with the $nca(u, v)$ nearest to the root (see Figure 4(a)). The information about the edge e is stored in variable mwe_u. If the parent x of the node u has the same information and the weight of the edge $w(u, x) > w(e)$ then the edge (u, v) is removed from the tree (see Figure 4(a-b) for the nodes 6 and 10). We use the red rule in an intensive way, because we remove all the edges with a weight upper than $w(e)$ in fundamental cycle of e.

This interpretation of the red rule allows to insure that after a recovering phase the remaining edges belong to a MST.

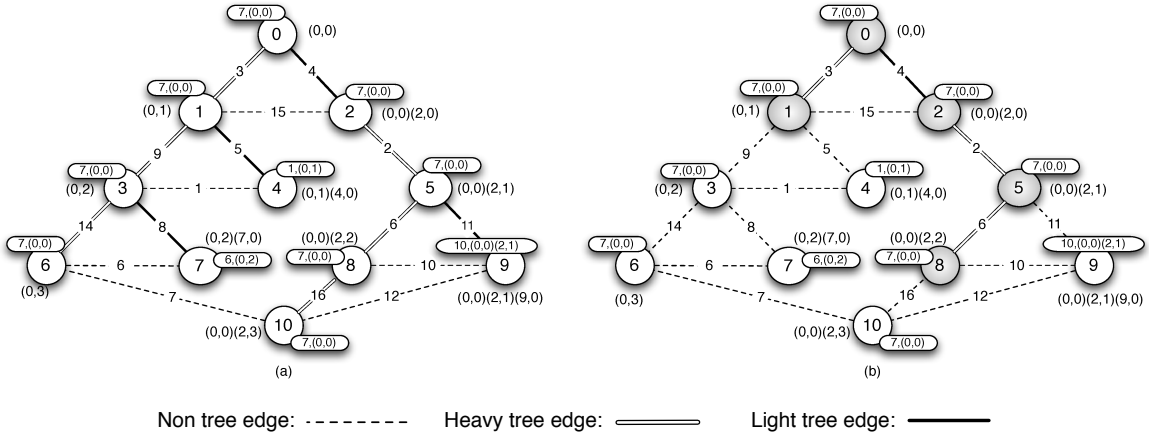

Figure 4: Minimum weighted edge computation and Tree correction. The bubble at each node v corresponds to the weight and the label of the common ancestor of the edge stored on variable mwe_v.

R$_{Min}$: [**Minimum computation**]
If $\neg$Cycle$(v) \wedge$ Label$(v) \wedge [(mwe_v \neq \text{MinEdge}(v) \wedge \text{MergeEdge}(v) \neq \emptyset) \vee (mwe_{p_v} = mwe_v \wedge \text{MergeEdge}(v) = \emptyset)]$
Then $mwe_v := \text{MinEdge}(v)$;

R$_{\lightning}$: [**MST Correction**]
If $\neg$Cycle$(v) \wedge$ Label$(v) \wedge$ NewFrag(v)
Then $p_v := \emptyset$; $\ell_v := (\mathsf{Id}_v, 0)$;

To summarize, in this section we explained how to compute the outgoing-edges and the fundamental cycles (Rule R$_{Min}$), and how to recover from a false tree (Rule R$_{\lightning}$). The next section addresses the fragments merging operation (Rules R$_{\triangleright\triangleleft}$ and R$_{\bigcirc}$).

3.2.2 Fragments merging

In this phase two rules are executed: R$_{\triangleright\triangleleft}$ and R$_{\bigcirc}$. Note that Rule R$_{Min}$ (described in the previous section) computes from the leaves to the root the minimum outgoing edge $e = (u, v)$ of the fragment F_u, with $u \in F_u$. The information about e are stored in the variable mwe, i.e., the weight of the edge and a common ancestor equal to ∞ to indicate that these information concern an outgoing edge. When a root r of F_u has stabilized its variable mwe_r, it starts a merging phase (Rule R$_{\triangleright\triangleleft}$). To this end, the nodes in the path between r and u are reoriented from r to v. During this reorientation the labels are locked. That is, each node x on the path between r and u (including r and excluding v) changes its label to: $\ell_v := (\perp, \perp)$. When a node u becomes the root of the fragment F_u it can merge with the fragment F_v. After the addition of the outgoing edge e, the labeling process is re-started (see Rule R$_{\bigcirc}$). The merging phase is repeated until a single fragment is obtained.

R$_{\triangleright\triangleleft}$: [**Merging**]
If NeedReorientation$(v) \wedge mwe_v =$ MergeEdge(v)

Then
If $(\exists u \in N(v) \backslash Child(v) \backslash \{p_v\}, w(u,v) = \text{MergeEdge}(v) \wedge \mathsf{Id}_v > \mathsf{Id}_u)$
Then
$p_v := \min\{\mathsf{Id}_u : u \in N(v) \backslash Child(v) \backslash \{p_v\} \wedge w(u,v) = \text{MergeEdge}(v)\}$;
$\ell_v := (\emptyset, \emptyset)$;
If $(\exists u \in N(v) \backslash Child(v) \backslash \{p_v\}, w(u,v) = \text{MergeEdge}(v) \wedge \mathsf{Id}_v < \mathsf{Id}_u \wedge p_u = \mathsf{Id}_v)$
Then $\ell_v := (\emptyset, \emptyset)$;
Else $p_v := \min\{\mathsf{Id}_u : u \in Child(v) \wedge mwe_v = \text{MergeEdge}(v)\}$;
$\ell_v := (\perp, \perp)$;

$R_{\bigcirc}$: [**End Merging**]
If $\neg$NeedReorientation$(v) \wedge$ EndReorientation(v)
Then $\ell_v := (\emptyset, \emptyset)$;

4 Correctness proof

Lemma 1 *Let $\mathcal{C}$ a configuration where the set of variables $p_v, v \in V$, form at least one cycle in the network. In a finite time, Algorithm* NCA-L *removes all the cycles from the network.*

Proof. If a node v has a parent which is not in its neighborhood or if v has no parent then the parent and the label variable of v is modified to $\emptyset$ and $(\mathsf{Id}_v, 0)$ respectively with Rule $R_{\odot}$.

A node v identifies a cycle with Predicate Cycle(v) which uses v's label and the label of its parent. In a legitimate configuration, v's label is smaller than the label of its parent and is constructed using the label of its parent, i.e., the label of the parent of v is included to v's label. Thus, if the label of v is included or is smaller than the label of its parent then a cycle is detected and Predicate Cycle(v) is true. In this case, v reinitiates its parent and label variable using Rule $R_{\odot}$.

In order to detect a cycle the label computation process must cross a part or all the nodes of the cycle. However, since we consider a distributed scheduler then all the nodes in a cycle can be activated and we can have a rotation of the labels of the nodes in the cycle. This may lead to a new configuration in which the labels cannot be used to detect a cycle, because the label of one node is not used to compute some other labels and to detect a cycle. To break this symmetry, we use the node identifiers with Predicate MinEnabled(v). This predicate allows to activate the node v iff v has no neighbor u such that u is activated and u's identifier is lower than v. Therefore, there is at least a node x in the cycle which is not activated and when the label of x is used by some other nodes to compute their labels then Predicate Cycle(x) is true and x breaks the cycle using Rule $R_{\odot}$. □

According to Lemma 1, if the system starts from a configuration which contains at least one cycle then all the cycles are removed from the network in a finite time. Therefore, in the following we consider only configurations containing no cycle.

Definition 2 (Legitimate state of NCA-L) *Let $\mathcal{C}$ a configuration with no cycle in the network, i.e., which contains a forest of trees $T = (V_T \subseteq V, E_T \subseteq E)$. The configuration $\mathcal{C}$ is legitimate for Algorithm* NCA-L *iff each node $v \in V_T$ satisfies one of the following conditions:*

1. *the label ℓ_v of v is equal to $\ell_{p_v}.(\mathsf{Id}_v, 0)$, if the edge between v and v's parent in T is a light edge or v is the root of the tree;*
2. *the label ℓ_v of v is equal to ℓ_{p_v} and $last(\ell_v)[1] = last(\ell_{parent})[1] + 1$, if the edge between v and v's parent in T is a heavy edge.*

Lemma 2 (Convergence for NCA-L) *Starting from an illegitimate configuration for Algorithm* NCA-L*, eventually Algorithm* NCA-L *reaches in a finite time a legitimate configuration.*

Proof. To compute correct node labels, heavy and light edges in each tree $T = (V_T, E_T)$ of the forest in the network must be identified. To this end, each node $v \in V_T$ maintains in the variable $size_v$ two information: the size of its subtree in T and the identifier of the child with the subtree of maximum number of nodes. Based on the first information given by its children $u \in V_T$ stored in variable $size_u$, each node v compute the size of its subtree in T and informs its child u if the edge $(u, v) \in E_T$ is a light or heavy edge. The edge $(u, v) \in E_T$ is a heavy edge if the second information stored in $size_v$ is equal to Id_u the identifier of u, a light edge otherwise. Therefore, each node $v \in V_T$ can detect if its label is correct according to its parent label. Note that Predicate MinEnabled(v) is used at node $v \in V$ to help to break cycle if we are in the case of a configuration described in proof of Lemma 1. Moreover, since we use the labeling scheme with minimum spanning tree computation rules Predicate TreeMerg(v) is used to forbid the label correction when it has been modified by Rule $\mathrm{R}_{\not\downarrow}, \mathrm{R}_{\rhd\lhd}$ and $\mathrm{R}_{\bigcirc}$.

The computation of the information stored in the variable $size_v$ at each node $v \in V_T$ is done via bottom-up fashion in the tree T. According to Predicate SizeC(v), if the variable $size_v$ is not equal to $(1, \perp)$ at a leaf node v in T then Predicate SizeC(v) $= false$ and v can execute Rule R_ℓ to correct its variable $size_v$. Otherwise according to Predicate SizeC(v), for any internal node $v \in V_T$ the first information of $size_v$ must be equal to one plus the sum of the size of the children subtrees and the second one to the identifier of its child with the maximum subtree size. Thus, if variable $size_v$ is not correct (i.e., Predicate SizeC(v) $= false$) then v can execute Rule R_ℓ to correct its variable $size_v$. Using the same argument, one can show by induction that for any internal node $v \in V_T$ we have $size_v = (1 + \sum_{u \in \mathrm{Child}(v)} size_u, \arg\max\{size_u : u \in \mathrm{Child}(v)\})$.

The computation of the node labels in a tree T is done via top-down fashion starting from the root of T. For convenience, a path is called *heavy* (resp. *light*) if it contains only heavy (resp. light) edges. Moreover, a node is called *heavy* (resp. *light*) if the edge between its parent and itself is a heavy (resp. light) edge. v is informed by its parent with $size_{p_v}$ if v is a heavy (i.e., $size_{p_v}[1] = \mathsf{Id}_v$) or light (i.e., $size_{p_v}[1] \neq \mathsf{Id}_v$) node. The root node v of T is also the root of a heavy path and its label must be equal to $(\mathsf{Id}_v, 0)$. According to Predicate Label(v) and $\mathrm{Label}_R(v)$, v can execute Rule R_ℓ to correct its label. Otherwise, we have two cases: heavy or light nodes. When the root have a correct label then all its children can compute their correct label. If a heavy (resp. light) node has a label different from ℓ_{p_v} and $last(\ell_v)[1] = last(\ell_{parent})[1] + 1$ (resp. $\ell_{p_v}.(\mathsf{Id}_v, 0)$) then we have Predicate $Heavy(v) = false$ (resp. Light(v) $= false$), $\mathrm{Label}_{Nd}(v) = false$ and Label(v) $= false$. Therefore, v can execute Rule R_ℓ to correct its label ℓ_v accordingly with its parent. Using the same argument, one can show by induction that for any internal node $v \in V_T$ we have ℓ_{p_v} and $last(\ell_v)[1] = last(\ell_{parent})[1] + 1$ (resp. $\ell_{p_v}.(\mathsf{Id}_v, 0)$) for heavy (resp. light) nodes. □

Lemma 3 (Closure for NCA-L) *The set of legitimate configurations for* NCA-L *is closed.*

Proof. According to Algorithm NCA-L, the labeling procedure is done using only Rule R_ℓ. In any legitimate configuration for Algorithm NCA-L, for any node $v \in V$ Predicate SizeC(v) and Label(v) are true and Rule R_ℓ cannot be executed by a node v. So, starting from a legitimate configuration for Algorithm NCA-L the system remains in a legitimate configuration. □

Definition 3 (Legitimate state of MST) *A configuration is legitimate for Algorithm* MST *iff each node $v \in V$ satisfies the following conditions:*

1. *a tree T spanning the set of nodes in V is constructed;*

2. T is of minimum weight among all spanning trees.

Lemma 4 *Let $T_i = (V_{T_i}, E_{T_i})$ a tree (or fragment). Eventually for each node $v \in V_{T_i}$ the variable mwe_v contains a pair of values: the weight of the minimum outgoing edge (u, v) of the fragment T_i and $\emptyset$, if a merging is possible between two fragments T_j and T_i, $(j \neq i)$.*

Proof. We assume that T_i and T_j, $(j \neq i)$ are two distinct coherent trees (i.e., different root and correct labels, otherwise Rule $R_{\odot}$ and R_{ℓ} are used to correct the trees) in the network and that a merging is possible between T_i and T_j. The computation of the minimum outgoing edge of T_i (resp. T_j) is done in a bottom-up fashion. We consider the tree T_i but the computation in T_j is done in a same way. A leaf node v can compute and store in its variable mwe_v its local adjacent minimum outgoing edge leading to another tree. Macro MinEdge(v) returns the local minimum outgoing edge if Macro MergeEdge(v) $\neq \emptyset$. To this end, if there is an adjacent outgoing edge (u, v) leading to another tree T_j (i.e., MergeAdj(v) $\neq \emptyset$ and MergeEdge(v) $\neq \emptyset$) and we have $mwe_v \neq$ MinEdge(v) then v can execute Rule R_{Min} to compute its local outgoing edge stored in the variable mwe_v. A internal node v must use the local outgoing edges computed by its children and selects the edge of minimum weight among these edges, then it compares this value with the weight of its adjacent local outgoing edge and again it holds the edge of minimum weight. The selection of its children minimum outgoing edge is done by Macro MergeChild(v) and the computation of its local outgoing edge is done by Macro MergeAdj(v) as for a leaf node. Thus, if there is an outgoing edge which can be used to make a merging with another tree (i.e., MergeEdge(v) $\neq \emptyset$) and we have $mwe_v \neq$ MinEdge(v) for a internal node v, then v can execute Rule R_{Min} to compute in the variable mwe_v its local minimum outgoing edge. Using the same argument, one can show by induction that for any internal node $v \in V_{T_i}$ we have $mwe_v =$ MergeEdge. Therefore, the local outgoing edge computed by the root node v is the minimum outgoing edge of T_i. □

Lemma 5 *Let $T_i = (V_{T_i}, E_{T_i})$ a tree (or fragment). Eventually for each node $v \in V_{T_i}$ the variable mwe_v contains a pair of values: the weight of an edge $(u, v) \notin E_{T_i}$ and the label of the nearest common ancestor of $u \in V_{T_i}$ and v. Moreover, eventually all local internal edges of v are computed.*

Proof. We assume that $T_i = (V_{T_i}, E_{T_i})$ is a coherent tree, otherwise Rule $R_{\odot}$ and R_{ℓ} are used to break the cycles and to correct the labels. Each node $v \in V_{T_i}$ starts to compute local internal edges (i.e., edges (x, y) such that $x, y \in V_{T_i}$ and $x = v$ or x or y is in the subtree of v) when it has no local outgoing edge (adjacent outgoing edge or outgoing edge given by a child) leading to another tree. In this case, a node v informs its parent of its local internal edges using its variable mwe_v. Macro RecoverEdge(v) returns the local internal edge of v which has the common ancestor nearest from the root among the internal edges that were not taken into account by its parent using the node labels. Each node $v \in V_{T_i}$ which is in a recover phase sends all its local internal edges. To this end, v compute its next local internal edge when its parent has taken into account v's current local internal edge (i.e., $mwe_{p_v} = mwe_v$). Thus, the information of internal edges are put back up in the tree until reaching the nearest common ancestor and v do not wait an acknowledgement from the nearest common ancestor to send its next internal edge. Moreover, Macro FarLca(v) compute the next internal edge adjacent to v such that the label of the nearest common ancestor associated to (u, v) with $u \in N(v)$ is greater (according to operator $>_{\ell}$) than $mwe_v[1]$ which is used to define an order on the internal edges. Otherwise, if FarLca(v) $= \emptyset$ then according to Macro RecoverAdj(v) the node v reset the computation of its adjacent internal edge to assure that every internal edge is taken into account. The recover phase is started at node v if v has no local outgoing edge (i.e., MergeEdge(v) $= \emptyset$) and v's

parent has taken into account the information associated to its current internal edge and stored in variable mwe_v (i.e., $mwe_{p_v} = mwe_v$). In this case, the guard of Rule R_{Min} is satisfied and v can execute Rule R_{Min} to update its variable mwe_v with the information of its next local internal edge. The information given by a child are stopped at the nearest common ancestor v of the corresponding internal edge since Macro FarLcaChild(v) selects only mwe_u from a child u such that $mwe_u[1] \leq_\ell \ell_v$. □

Lemma 6 *Let $T = (V_T, E_T)$ a tree and any edge $(x, y) \in E_T$. Eventually, if (x, y) is not part of a minimum spanning tree of the network then (x, y) is removed from T.*

Proof. We assume that $T = (V_T, E_T)$ is a coherent tree, otherwise Rule $R_\odot$ and R_ℓ are used to break the cycles and to correct node labels. Let an edge $(x, y) \in E_T$ (w.l.o.g. $p_y = x$) which is not part of a minimum spanning tree. As the network has a finite size then there is a time after which there exists no merging between two trees in the network. Thus according to Lemma 5 each internal edge e of T is put back up in T until reaching the nearest common ancestor associated to e. Since (x, y) is not in a minimum spanning tree, there is an edge (u, v) such that $w(u, v) < w(x, y)$ and (x, y) is on the path between u and $nca(u, v)$ or between v and $nca(u, v)$. So, there is a time such that $mwe_y = (w(u, v), nca(u, v))$ and $mwe_x = (w(u, v), nca(u, v))$ according to Lemma 5. Then Predicate NewFrag(y) returns true because we have $mwe_y = mwe_x, mwe_y[1] \neq \mathsf{Id}_y$ and $w(x, y) > w(u, v)$. Therefore, y can execute Rule $R_{\not\iota}$ to create a new tree rooted at y and as a consequence the edge (x, y) is removed from T. □

Lemma 7 *Let two distinct trees $T_i = (V_{T_i}, E_{T_i})$ and $T_j = (V_{T_j}, E_{T_j})$ with $i \neq j$. Let an edge (x, y) such that (x, y) is part of a minimum spanning tree and $x \in T_j$ and $y \in T_i$. Eventually (x, y) is used to merge the trees T_i and T_j.*

Proof. We assume that $T_i = (V_{T_i}, E_{T_i})$ and $T_j = (V_{T_j}, E_{T_j})$ are coherent trees, otherwise Rule $R_\odot$ and R_ℓ are used to break the cycles and to correct node labels. According to Lemma 4, the merging edge (x, y) is computed by the root and it starts the merging phase since only the root can choose the edge of minimum weight leading to another tree to use in order to make a merging. In the remainder, we focus on tree T_i but the same arguments are also true for T_j.

When the root v has finished to compute its minimum outgoing edge from its fragment (i.e., we have $mwe_v = \text{MergeEdge}(v)$) then v can execute Rule $R_{\triangleright\triangleleft}$ because Predicate NeedReorientation(v) is satisfied since v has a coherent label. Note that we permit the creation of cycles of length two only if at least one node has a label equal to $(\perp, \perp)$. Indeed, during the merging phase the orientation is reversed on the path between the root of T_i and the node adjacent to the edge used for the merging, that is why a cycle is detected in Rule $R_\odot$ if Predicate NeedReorientation(v) is not satisfied. Thus, v can change its variables p_v and ℓ_v as following. If there is an edge (x, y) adjacent to v such that $w(x, y) = mwe_v[0]$ and $y = v$ then v selects x as its new parent (only if $\mathsf{Id}_v > \mathsf{Id}_x$) and v changes its label to $(\emptyset, \emptyset)$ to informs its subtree that the merging is done. Otherwise, v selects its child u such that $mwe_u = mwe_v$ as its new parent and v changes its label to $(\perp, \perp)$ to inform u that a merging is started.
Any other node v on the path between the root and the node adjacent to the merging edge take part in the merging phase when its parent has selected v as its new parent (i.e., $p_{p_v} = \mathsf{Id}_v$) and v's parent label is equal to $(\perp, \perp)$. Thus, Predicate NeedReorientation(v) is satisfied and v can execute Rule $R_{\triangleright\triangleleft}$ since $mwe_v = \text{MergeEdge}(v)$ (otherwise Rule R_{Min} is executed to update its variable mwe_v). So, v changes its variables p_v and ℓ_v as described above for the root. Since the merging phase is done on a path, using the same argument one can show by induction that for any internal node $v \in V_{T_i}$ on the path between the root and the node y adjacent to the merging edge (except for y) we have $p_v = \min\{\mathsf{Id}_u : u \in Child(v) \wedge mwe_v = \text{MergeEdge}(v)\}$ and $\ell_v = (\perp, \perp)$ and for the node y we have $p_y = x$ and $\ell_y = (\emptyset, \emptyset)$. □

Lemma 8 *Eventually all the nodes have a correct label in the new fragment resulting from a merging phase.*

Proof. According to Lemma 7, a merging phase is done using the minimum outgoing edge (x, y) between two distinct trees T_i and T_j if it is possible. Moreover, when the edge (x, y) is added by the extremity of minimum identifier, w.l.o.g. let $y \in$, then y's label is equal to $(\emptyset, \emptyset)$ and the end of the merging phase is propagated in the resulting fragment T. In the reminder we focus on tree T_i but the same arguments are true for tree T_j.

Let the node $v \in T_i$ such that $p_v = y$ and $\ell_v = (\perp, \perp)$ (i.e., v is the child of y on the path between y and the old root of T_i). Predicate NeedReorientation(v) is false because v is not a root node and the label of its parent y is not equal to $(\perp, \perp)$. Moreover, Predicate EndReorientation(v) is true since y's label is equal to $(\emptyset, \emptyset)$ and v's label to $(\perp, \perp)$. Thus, v can execute Rule $\mathrm{R}_{\bigcirc}$ to modify its label to $(\emptyset, \emptyset)$. Using the same argument, one can show by induction that every node v on the path between y and the old root of T_i can execute Rule $\mathrm{R}_{\bigcirc}$, thus there is a time such that we have $\ell_v = (\emptyset, \emptyset)$.
Now we show that the other nodes in T_i can execute Rule $\mathrm{R}_{\bigcirc}$. Consider the node $r \in V_{T_i}$ such that r is the old root of T_i and $\ell_z = (\emptyset, \emptyset)$. Let a node $v \in V_{T_i}$ such that $p_v = \mathsf{Id}_z$. Predicate NeedReorientation(v) is false because v is not a root node and the label of its parent is not equal to $(\perp, \perp)$. Moreover, Predicate EndReorientation(v) is true because $\ell_z = (\emptyset, \emptyset)$ and v's label is not equal to $(\perp, \perp)$ since v is not on the path between y and the old root of T_i, and v's label is different from $(\emptyset, \emptyset)$. Note that since the start of the merging phase, Predicate TreeMerg(v) is true because Predicate NeedReorientation(v) or EndReorientation(v) is true. So, Rule R_ℓ cannot be executed by v and v's label has not changed. Thus, v can execute Rule $\mathrm{R}_{\bigcirc}$ to modify its label to $(\emptyset, \emptyset)$. Using the same argument, one can show by induction that every node v on the path between z and a leaf node can execute Rule $\mathrm{R}_{\bigcirc}$, thus there is a time such that we have $\ell_v = (\emptyset, \emptyset)$.

Every node v in the resulting fragment T can execute Rule R_ℓ when the edge (x, y) is added in T and y's label has been modified from $(\emptyset, \emptyset)$ to its new label based on x's label. Indeed, in this case for every node v, with $v \neq y$ and $v \in V_{T_i}$, Predicate TreeMerg(v) is false and v can execute Rule R_ℓ. Therefore, there is a time such that every node v in T has a correct label. □

Lemma 9 (Convergence for `MST`) *Starting from an illegitimate configuration for Algorithm* `MST`*, eventually Algorithm* `MST` *reaches in a finite time a legitimate configuration.*

Proof. We assume that there is a forest of trees $T_i, 1 \leq i \leq n$, in the network, otherwise according to Lemma 1 Rule $\mathrm{R}_{\odot}$ is executed to remove the cycle from the network. Moreover, we assume also that the node's label are correct in tree T_i, otherwise according to Lemma 2 and 3 there is a time such that the node's label are corrected.

According to Lemma 5 and 6, if an edge $(u, v) \in E_{T_i}$ and (u, v) is part of no minimum spanning tree of the network then (u, v) is removed from tree T_i. Thus, there is a time such that the existing fragments in the network are part of a minimum spanning tree. According to Lemmas 4 and 7, eventually if there are at least two distinct fragments then a merging phase is started. Moreover, node labels are corrected after a merging phase according to Lemma 8. Since the size of the network is finite there is a finite number of merging. Therefore, in a finite time a spanning tree of minimum weight is computed by Algorithm `MST`. □

Lemma 10 (Closure for `MST`) *The set of legitimate configurations for* `MST` *is closed.*

Proof. Let $\mathcal{C}$ a legitimate configuration such $T = (V_T, E_T)$ is a minimum spanning tree of the network and an edge $(x, y) \in E_T$. To be illegitimate, the configuration $\mathcal{C}$ must contain an

edge (x, y) such that it exists an edge $e = (u, v)$ with $w(u, v) < w(x, y)$ and (x, y) and (u, v) are included in the same fundamental cycle C_e. Thus, this imply that the edge e is not used to verify if it is possible to replace an edge of C_e with (u, v) which contradicts Lemmas 5 and 6. Moreover, since T is a spanning tree then no merging is done in the network. Therefore, starting from a legitimate configuration for Algorithm `MST` a legitimate configuration is preserved. □

5 Complexity proofs

In the following we discuss the complexity issues of our solution.

Lemma 11 *Algorithms* `NCA-L` *and* `MST` *have a space complexity of* $O(\log^2 n)$ *bits.*

Proof. Algorithm `NCA-L` uses three variables : $p_v, \ell_v, size_v$. The first and the last one are respectively a pointer to a neighbor node and a pair of integers, each one needs $O(\log n)$ bits. However, the variable ℓ_v is a list of pairs of integers. A new pair of integers is added to the list when a light edge is created in the tree. As noticed in [1], there are at most $\log n$ light edges on the path from a leaf to the root, i.e., at most $\log n$ pairs of integers. Thus, the variable ℓ_v uses $\log n \times \log n$ bits.

Algorithm `MST` uses an additional variable mwe_v which is a pair composed of an integer and the label of a node. The label of a node is stored in variable ℓ_v which uses $\log^2 n$ bits. Thus, the variable mwe_v needs $\log^2 n$ bits.

Therefore, Algorithms `NCA-L` and `MST` use $O(\log^2 n)$ bits of memory at each node. □

Lemma 12 *Starting from any configuration, all cycles are removed from the network in at most* $O(n^2)$ *rounds, with* n *the number of nodes in the network.*

Proof. As explained in the proof of Lemma 1, to break a cycle C_k a part of the nodes in C_k must compute their new labels, that is a label computation must be initiated from one node and then this process must cross C_k. Thus, the worst case is a configuration in which all the nodes in C_k have to compute their new labels using Rule R_ℓ to detect the presence of cycle C_k. Therefore, at most $O(n)$ rounds are needed to compute the new label of the nodes in C_k based on the label of one node x in C_k. According to Lemma 1, when this computation is done the cycle C_k is detected and removed by the node x. At most $O(n)$ additional rounds are needed to break the cycle C_k.

Since there is at most $n/2$ cycles in a network, at most $O(n^2)$ rounds are needed to remove all the cycles from the network. □

Lemma 13 *Starting from a configuration which contains a tree* T*, using Algorithm* `NCA-L` *any node* $v \in V_T$ *has a correct label in at most* $O(n)$ *rounds.*

Proof. As described in proof of Lemma 2, the correction of node labels is done using a bottom-up computation followed by a top-down computation in the tree $T = (V_T, E_T)$.

The bottom-up computation is started by the leaves of T, when leaf nodes $v \in V_T$ have corrected their variable $size_v$ to $(1, \perp)$ then internal nodes $u \in V_T$ can start to correct their variable $size_u$. An internal node $v \in V_T$ computes a correct value in its variable $size_v$ using Rule R_ℓ when all its children u have a correct value in their variable $size_u$. Since the computation is done in a tree sub-graph then in at most $O(n)$ rounds each node $v \in V_T$ has corrected its variable $size_v$.

The top-down computation is started by the root of the tree T. When the root v has a correct value in variable $size_v$ then the computation of correct labels can start. Thus, if the

parent of a node v has a correct value in its variable $size_{p_v}$ and ℓ_{p_v} then v can compute its correct label in ℓ_v using Rule R_ℓ. As for the bottom-up computation, the top-down computation is done in at most $O(n)$ rounds since it is performed in a tree sub-graph.

Therefore, in at most $O(n)$ rounds each node v in the tree T has a correct label stored in variable $size_v$. □

Lemma 14 *Starting from any configuration, Algorithm* `NCA-L` *reaches a legitimate configuration in at most* $O(n^2)$ *rounds.*

Proof. The initial configuration $\mathcal{C}$ could contain one or more cycles, so according to Lemma 12 in at most $O(n^2)$ rounds the system reaches a new configuration $\mathcal{C}'$ which contains no cycle. Moreover according to Lemma 13, the nodes v in each tree T in the configuration $\mathcal{C}'$ have a correct label in at most $O(n)$ rounds. Therefore, starting from an arbitrary configuration each node $v \in V$ computes its correct label in at most $O(n^2)$ rounds. □

Lemma 15 *Starting from any configuration, Algorithm* `MST` *reaches a legitimate configuration in at most* $O(n^2)$ *rounds.*

Proof. According to Lemma 12, starting from any configuration after at most $O(n^2)$ rounds all the cycles are removed from the network, i.e., it remains a forest of trees after at most $O(n^2)$ rounds. Moreover, according to Lemma 13 in at most $O(n)$ additional rounds each node $v \in V$ has a correct label since each node belongs to a unique tree.

According to the description of Algorithm `MST`, Macro MinEdge(v) and Lemma 5, when it is possible to make a merging between two distinct trees in the forest a merging phase is started. This merging phase is done in three steps: (1) information corresponding to the minimum outgoing edge is propagated in a bottom-up fashion in each tree, (2) the orientation is reversed from the root of a tree until reaching the node in the tree adjacent to the minimum outgoing edge, and (3) the node labels are changed to inform of the end of the merging phase, followed by a propagation of the new correct node labels in the new tree resulting from the merging phase.

The first step is a propagation of information in a bottom-up fashion in a tree which is done in at most $O(n)$ rounds. The second step reverses and propagates new node labels on a part of the tree (between the root and the node adjacent to the minimum outgoing edge) which is done in at most $O(n)$ rounds too. Step 3 modifies the label of the nodes which have changed their parent pointer in step 2, so this last step takes also at most $O(n)$ rounds and the relabeling of the nodes in the new tree is done in at most $O(n)$ rounds according to Lemma 13. Thus, a merging phase is accomplished in at most $O(n)$ rounds and as there are in the worst case n trees then in at most $O(n^2)$ rounds a spanning tree is constructed.

When there is no possible merging for a given fragment (or tree) T_i in the forest then the correction phase concerning T_i is started. In a tree T_i, the internal edges (i.e., whose two endpoints are in T_i) are sent upward in T_i in order to detect incorrect tree edges. The internal edges e are sent following an order on the distance between the common ancestor $nca(e)$ and the root of T_i, by sending first the edge e with the nearest common ancestor $nca(e)$ from the root. Let $h(T_i)$ be the height of tree T_i and $d(v)$ be the distance from $v \in T_i$ to the root of T_i. Thus, an internal (resp. leaf) node has at most $d(v)-2$ (resp. $d(v)-1$) adjacent internal edges. Since a leaf node could have a lower priority (compared to its ancestors) to send all its adjacent internal edges, then the worst case to correct a tree is the case of a chain. Indeed, if the last internal edge of a leaf node x must be used to detect an incorrect tree edge then x may have to wait that all its ancestors in the chain have sent their internal edges of higher priority. Thus, starting from any configuration after at most $O(h(T_i)^2)$ rounds T_i contains no incorrect edges. Note that this is the worst case time to detect the farthest incorrect tree edge from the root

of T_i, otherwise the correction phase is stopped earlier for nearest incorrect tree edges because the merging phase has a higher priority than the correction phase. Moreover, after $O(h(T_i)^2)$ rounds all the new edges used by T_i for a merging are correct tree edges for T_i. So, T_i does not remove another tree edge in a new correction phase. Hence starting from any configuration, a correction phase deletes all the incorrect tree edges of a spanning tree after at most $O(n^2)$ rounds and no new tree edges are removed by a correction phase.

Therefore, starting from an arbitrary configuration Algorithm `MST` constructs a minimum spanning tree in at most $O(n^2)$ rounds. □

6 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 logarithmic memory in the size of the network, it scales well since it does not rely on any global parameter of the system, it is fast — its time complexity is the better known in self-stabilizing settings. Additionally, it self-recovers from any transient fault. The time complexity is $O(n^2)$ rounds and the space complexity is $O(log^2 n)$.

References

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

[2] Lélia Blin, Maria Potop-Butucaru, Stephane Rovedakis, Sébastien Tixeuil. A New Self-stabilizing Minimum Spanning Tree Construction with Loop-Free Property. *DISC*, volume 5805 of *Lecture Notes in Computer Science*, pages 407–422. Springer 2009.

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

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

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

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

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

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

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

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

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

[13] 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.

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

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