# Maintaining a Spanning Forest in Highly Dynamic Networks: The Synchronous Case

Matthieu Barjon, Arnaud Casteigts, Serge Chaumette,
Colette Johnen, and Yessin M. Neggaz

LaBRI, University of Bordeaux

**Abstract.** Highly dynamic networks are characterized by frequent changes in the availability of communication links. Many of these networks are in general partitioned into several components that keep splitting and merging continuously and unpredictably. We present an algorithm that strives to maintain a forest of spanning trees in such networks, without any kind of assumption on the rate of changes. Our algorithm is the adaptation of a coarse-grain interaction algorithm (Casteigts et al., 2013) to the synchronous message passing model (for dynamic networks). While the high-level principles of the coarse-grain variant are preserved, the new algorithm turns out to be significantly more complex. In particular, it involves a new technique that consists of maintaining a distributed permutation of the set of all nodes ids throughout the execution. The algorithm also inherits the properties of its original variant: It relies on purely localized decisions, for which no global information is ever collected at the nodes, and yet it maintains a number of critical properties whatever the frequency and scale of the changes are. In particular, the network remains always covered by a spanning forest in which 1) no cycle can ever appear, 2) every node belongs to a tree, and 3) after an arbitrary number of edge disappearance, all maximal subtrees immediately restore exactly one token (the root). These properties are ensured whatever the dynamics, even if it is sustained for an arbitrary long period of time. Optimality is not the focus here, however convergence to a single tree per component eventually occurs if the network stops changing (which is never expected to happen, though). The algorithm correctness is proven and its behavior is tested in real world scenario through experimentations.

## 1 Introduction

The current development of mobile and wireless technologies enables direct *ad hoc* communication between various kinds of mobile entities, such as vehicles, smartphones, terrestrian robots, flying robots, or satellites. In all these contexts, the set of communication links depends on distances between entities, thus the network topology changes continuously as the entities move. Not only changes are frequent, but in general they even make the network partitionned. Clearly, the usual assumption of connectivity does not hold here, although another form

of connectivity is often available over time and space (temporal connectivity). Also, the classical view of a network whose dynamics corresponds to *failures* is no longer suitable in these scenarios, where dynamics is the norm rather than the exception.

This induces a shift in paradigm that strongly impacts algorithms. In fact, it even impacts the problems themselves. What does it mean, for instance, to elect a leader in such a network? Is the objective to distinguish a unique leader network wide, whose leadership then takes place over time and space, or is it to *maintain* a leader in each connected component, so that the decisions concerning each component are taken quickly and locally. The same remark holds for spanning trees. Should an algorithm construct a unique, global tree whose logical edges survive intermittence, or should it build and maintain a *forest* of trees that strive to cover collectively all components in each instant? Both viewpoints make sense, and so far, were little studied in distributed computing (see e.g. [4,11] for temporal trees, [3,10] for maintained trees).

We focus on the second interpretation, which reflects a variety of scenarios where the expected output of the algorithm should relate to the *immediate* configuration (*e.g.* direct social networking, swarming of flying robots, vehicles platooning on the road). A particular feature of this type of algorithms is that they never terminate. More significantly, in highly dynamic networks, they are not even expected to stabilize to an optimal state (say, a single tree per component), unless the changes stop, which never happens. This precludes, in particular, all approaches whereby the computation of a new solution requires the previous computation to have completed.

This paper is an attempt to understand what can still be computed (and guaranteed) when no assumptions are made on the network dynamics: neither on the rate of change, nor on their simultaneity, nor on global connectivity. In other words, the topology is controlled by an almighty adversary. In this seemingly chaotic context, we present an algorithm that strives to maintain as few trees per components as possible, while always guaranteeing some properties.

### 1.1 Related work

Several works have addressed the spanning tree problem in dynamic networks, with different goals and assumptions. Burman and Kutten [8] and Kravchik and Kutten [13] consider a self-stabilizing approach where the legal state corresponds to having a (single) minimum spanning tree and the faults are topological changes. The strategy consists in recomputing the entire tree whenever changes occur. This general approach, sometimes called the "blast away" approach, is meaningful if stable periods of time exist, which is not assumed here.

Many spanning tree algorithms rely on random walks for their elegance and simplicity, as well as for the inherent localized paradigm they offer. In particular, approaches that involve multiple coalescing random walks allow for uniform initialization (each node starts with the same state) and topology independence (same strategy whatever the graph). Pionneering studies involving such processes include Bar-Ilan and Zernik [6] (for the problem of election and spanning

tree), Israeli and Jalfon [12] (mutual exclusion), and Chapter 14 of Aldous and Fill [2] (for general analysis).

The principle of using coalescing random walks to build spanning trees in mildly dynamic networks was used by Baala et al. [1] and Abbas et al. [5], where tokens are annexing territories gradually by capturing each other. Regarding dynamicity, both algorithms require the nodes to know an upper bound on the cover time of the random walk, in order to regenerate a token if they are not visited during a long-enough period of time. Besides the strength of this assumption (akin to knowing the number of nodes $n$, or the size of components in our case), the efficiency of the timeout approach decreases dramatically with the rate of topological changes. In particular, if they are more frequent than the cover time (itself in $O(n^3)$), then the tree is constantly fragmented into "dead" pieces that lack a root, and thus a leader.

Another algorithm based on random walks is proposed by Bernard et al. [7]. Here, the tree is constantly redefined as the token moves (in a way that reminds the snake game). Since the token moves only over present edges, those edges that have disappeared are naturally cleaned out of the tree as the walk proceeds. Hence, the algorithm can tolerate failure of the tree edges. However it still suffers from detecting the disappearance of tokens using timeouts based on the cover time, which as we have seen, suits only slow dynamics.

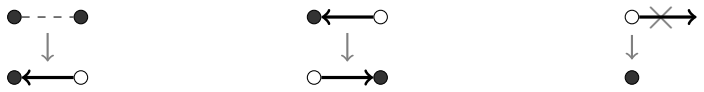
A recent work by Awerbuch et al. [3] addresses the maintenance of *minimum* spanning trees in dynamic networks. The paper shows that a solution to the problem can be updated after a topological change using $O(n)$ messages (and same time), while the $O(m)$ messages of the "blast away" approach was thought to be optimal. (This demonstrates, incidentally, the revelance of *updating* a solution rather than recomputing it from scratch in the case of minimum spanning trees.) The algorithm has good properties for highly dynamic networks. For instance, it considers as natural the fact that components may split or merge perpetually. Furthermore, it tolerates new topological events while an ongoing update operation is executing. In this case, update operations are enqueued and consistently executed one after the other. While this mechanism allows for an arbitrary number of topological events *at times*, it still requires that such burst of changes are only episodical and that the network remains eventually stable for (at least) a linear amount of time in the number of nodes, in order for the update operations to complete and thus the logical tree to be consistent with physical reality.

All the aforementioned algorithms either assume that *global update* operations (e.g. wave mechanisms) can instantly or eventually be performed, or that some node can collect *global information* about the tree structure. As far as dynamics is concerned, this forbids arbitrary and ever going changes to occur in the network.

### 1.2 The spanning forest principle.

A purely localized scheme was proposed by Casteigts et al. [10] for the maintenance of a (non-minimum) spanning forest in unrestricted dynamic networks, us-

ing a coarse grain interaction model inspired from graph relabeling systems [15]. It can be described informally as follows. Initially every node hosts a token and is the *root* of its own individual tree. Whenever two roots arrive at the endpoints of a same edge (see merging rule on Figure 1), one of them destroys its tokens and select the other as parent (*i.e.* the trees are merged). The rest of the time, each token executes a random walk within its own tree in the search for other merging opportunities (circulation rule). Tree relations are flipped accordingly. The fact that the random walk is *confined* to the underlying tree is crucial and different from all algorithms discussed above, in which they were free to roam everywhere without restriction. This simple feature induces very attractive properties for highly dynamic networks. In particular, whenever an edge of the tree disappears, the child side of that edge knows instantly that no token remains on its whole subtree. It can thus regenerate a token (*i.e.* become root) *instantly*, without global concertation nor further information collection. As a result, both merging and splitting of trees are managed in a purely localized fashion.



(a) Merging rule (b) Circulation rule (c) Regeneration rule

**Fig. 1.** Spanning forest principle (high-level representation). *Black nodes are those having a token. Black directed edges denote child-to-parent relationships. Gray vertical arrows represent transitions.*

At an abstract graph level, this very simple scheme guarantees that the network remains covered by a spanning forest at any time, in which 1) no cycle can ever appear, 2) maximal subtrees are always directed rooted trees (with a token at the root), and 3) every node always belongs to such a tree, whatever the chaos of topological changes are. On the other hand, it is not expected to reach an optimal state where a single tree covers each connected component. Even if the network were to stabilize, convergence to the optimum (though easy to be made certain) would not be expected to occur fast. Whether this general principle could be implemented in a message passing model remained an open question.

### 1.3 Our contribution.

This paper provides an implementation of the spanning forest principle in the synchronous message-passing model. Due to the loss of atomicity and exclusivity in the interaction, the algorithm turns out to be much more sophisticated than its original counterpart, while still reflecting the very same high-level principle. In particular, it involves the use of an original technique (which we refer to as the *unique score* technique) that consists of maintaining, network-wide, a set of

`score` variables (one for each node) that remain a permutation of the set of all ids. This mechanism allows us to break symmetry and avoid the formation of cycle in a context where ids alone could not. The paper is organized as follows. In Section 2, we present the model and notations that we use throughout the paper. Then Section 3 presents the algorithm and Section 4 establishes its correctness. Section 5 presents the validation of our algorithm experimentally.

## 2 Model and notations

The network is represented by an untimed evolving graph $\mathcal{G} = (G_1, G_2, \dots)$, such that $G_i = (V, E_i)$, where $V$ is a static set of vertices and $E_i$ is a dynamically changing set of undirected edges. Following Kuhn et al. [14], we consider a synchronous (thus rounded) computational model, where in each round $i$, the adversary chooses the set of edges $E_i$ that are present. In our case, this set is arbitrary (*i.e.* the adversary is unrestricted). At the beginning of each round, each node sends a message that it has prepared at the end of the previous round. This message is sent to all its neighbors in $E_i$, although the list of these neighbors is not know by the node. Then it receives all messages sent by its neighbors (in the same round), and finally computes its new state and the next message. Hence, each round corresponds to three phases `(send, receive, compute)`, which corresponds to a rotation of the original model of [14] where the phases are `(compute, send, receive)`. This adaptation is not necessary, but it allows us to formulate correctness of our algorithm in terms of the state of the nodes *after* each round rather than in the middle of rounds.

We assume that the nodes have a unique identifier taken from a totally ordered set, that is, for any two nodes $u$ and $v$, it either holds that $id(u) > id(v)$ or $id(v) < id(u)$. A node can specify what neighbor its message is intended to (although all neighbors will receive it) by setting the `target` field of that message. Symmetrically, the *id* of the emitter of a message can be read in the `sender` field of that message. Since the edges are undirected, if $u$ receives a message from $v$ at round $i$, then $v$ also receives a message from $u$ at that round. We call this property the *reciprocity principle* and it is an important ingredient for the correctness of our algorithm.

Using synchronous rounds allows us to represent the progress of the execution as a sequence of *configurations* $(C_0, C_1, C_2, ..., C_i)$, where each $C_i$ corresponds to the state of the system *after* round $i$ (except for $C_0$, the initial state). Each *configuration* consists of the union of all nodes variables, defined next.

## 3 The Spanning Forest Algorithm

### 3.1 State variables

Besides the `id` variable, which we assume is externally initialized, each node has a set of variable that reflects its situation in the tree: `status` accounts for the possession of a token (`T` if it has a token, `N` if it does not); `parent` contains the `id`

of this node's parent ($\perp$ if it has none); `children` contains the set of this node's children ($\emptyset$ if it has none). Observe that both variables `status` and `parent` are somewhat redundant, since in the spanning forest principle (see Section 1.2) the possession of a token is equivalent to being a root. Our algorithm enforces this equivalence, yet, keeping both variables separated simplifies the description of the algorithm and our ability to think of it intuitively. Variable `neighbors` contains the set of nodes from which a message was received in the last reception. These neighbors may or may not belong to the same tree as the current node. Variable `contender` contains the `id` of a neighbor that the current node considers selecting as parent in the next round (or $\perp$ if there is no such node). Finally, the variable `score` is the main ingredient of our cycle-avoidance mechanism, whose role is described below.

**Initial values:** All the nodes are uniformly initialized. They are initially the root of their own individual tree (*i.e.* `status` $= T$, `parent` $= \perp$, and `children` $= \emptyset$). They know none of their neighbors (`neighbors` $= \emptyset$), have no contenders (`contenders` $= \perp$), and their `score` is set to their own `id`.

### 3.2 Structure of a message (and associated variables)

Messages are composed of a number of fields: `sender` is the id of the sending node; `senderStatus` its status (either `T` or `N`); and `score` its score when the message was prepared. The field `action` is one of $\{FLIP, SELECT, HELLO\}$. Informally, $SELECT$ messages are sent by a root node to another root node to signify that it "adopts" it as a parent (merging operation); $FLIP$ messages are sent by a root node to circulate the token to one of its children (circulation operation); $HELLO$ messages are sent by a node by default, when none of the other messages are sent, to make its presence and status known by its neighbors. Finally, `target` is the id of the neighbor to which a FLIP or a SELECT message are intended ($\perp$ for HELLO messages).

Received messages are stored in a variable `mailbox`, which is a map collection whose *keys* are the senders id (*i.e.,* a message whose sender id is $u$ can be accessed as `mailbox[`$u$`]`). In each round, the algorithm makes use of a `RECEIVE()` function that clears the mailbox and fill it with all the messages received in that round (one for each physical neighbor). A node can thus update the set of its neighbors by fetching the *keys* of its mailbox. Similarly, it can eliminate from its list of children those nodes which are no more neighbor.

As mentioned above, every node prepares at the end of a round the message to be sent at the beginning of the next round. This message is stored in a variable `outMessage`. We allow the short hand `m` $\leftarrow (a, b, c, d, e)$ to define a new message $m$ whose emitter is node $a$ (with status $b$ and score $e$); target is node $d$; and action is $c$.

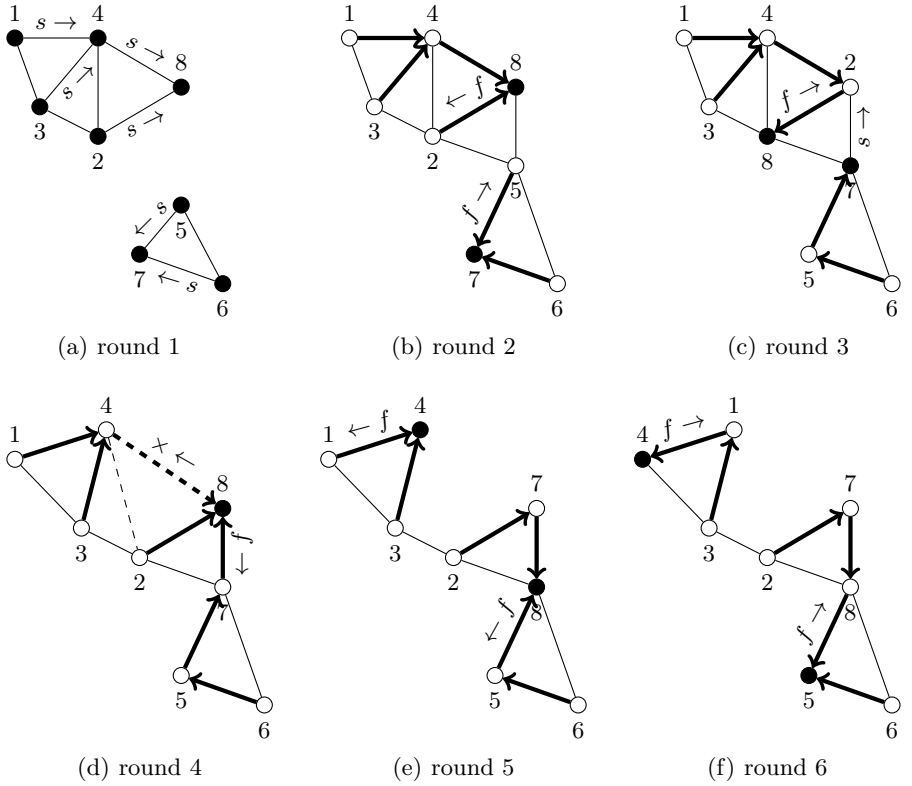
**Initial values:** The mailbox is initially empty (`mailbox` $= \emptyset$) and `outMessage` is initialized to $(id, T, HELLO, \perp, id)$.

### 3.3 Informal description of the algorithm

The algorithm implements the general scheme presented in Section 1.2. In this Section we explain how each of the three core operations (*merging*, *circulation*, *regeneration*) is implemented. Then we discuss the specificities of the merging operation in more detail and the problems that arise due to its entanglement with the circulation operation, a fact due to the loss of atomicity in the message passing model. The resulting solution is substantially more sophisticated than its original scheme, and yet it faithfully reflects the same high-level principle. Let us start with some generalities. In each round, each node broadcasts to its neighbors a message containing, among others, its status (`T` or `N`) and an action (SELECT, FLIP, or HELLO). Whether or not the message is intended to a specific *target* (which is the case for SELECT and FLIP messages), all the nodes who receive it can possibly use this information for their own decisions. More generally, based on the received information and the local state, each node computes at the end of the round its new status and the local structure of its tree (variables `children` and `parent`), then it prepares the next message to be sent. We know describe the three operations. Throughout the explanations, the reader is invited to refer to Figure 2, where an example of execution involving all of them is shown. All details are also given in the listings of Algorithm 1 and 2.

**Merging:** If a root (*i.e.* a node having a token), say $v$, detects the existence of a neighbor root with higher `score` than its own, then it considers that node as a possible `contender`, *i.e.* as a node that it might select as a parent in the next round. If several such roots exist, then the one with highest score, say $u$, is chosen. At the beginning of the next round, $v$ sends a $SELECT$ message to $u$ to inform it that it is its new parent. Two cases are possible: either the considered edge is still present in that round, or it disappeared in-between both rounds. If it is still present, then $u$ receives the message and adds $v$ to its children list, among others (Line 16). As for $v$, it sets its `parent` variable to $u$ and its `status` to N (Lines 8 and 9). It the edge disappeared, then $u$ does not receive the message, which is lost. However, due to the reciprocity of message exchange, $v$ does not receive a message from $u$ either and thus simply does not executes the corresponding changes. By the end of the round, either the trees are properly merged, or they are properly separated.

**Circulation:** If a root $v$ does not detect another root with higher score, then it selects one of its children at random, if it has any (see Line 27), otherwise it simply remains root. Randomness is not a strict requirement of our algorithm and replacing it with any deterministic strategy would not affect correctness of the algorithm. Once the child is chosen, say $u$, the root prepares a FLIP message intended to $u$, and sends it at the beginning of the next round. Two cases are again possible, whether or not the edge $\{u, v\}$ is still present in that round. If it is still present, then $u$ receives the message, it updates its status and adds $v$ to its children list, among others (Lines 15 and Line 16). As for $v$, it sets its `parent`




(a) round 1 (b) round 2 (c) round 3

(d) round 4 (e) round 5 (f) round 6

**Fig. 2.** Example of execution of the algorithm which illustrates all types of operations: parent selection ($s \rightarrow$), token circulation ($f \rightarrow$), and tree disconnection ($\times \leftarrow$). *The first two symbols represent FLIP or SELECT messages to be sent in the next round. Black (resp. white) nodes are those (not) having a token at the beginning of the round. Tree edges are represented by bold directed edges. Dash edges have just disappeared.*

variable to $u$ and its `status` to N (Lines 8 and 9). If the edge disappeared, then $v$ can detect it as before simply does not executes the corresponding changes. Node $u$, on the other hand, detects that the edge leading to its current parent disappeared, thus it regenerates a token (discussed next). Notice that in the absence of a merging opportunity, a node receiving the token in round $i$ will immediately prepare a FLIP message to circulate the token in the next round. Unless the tree is composed of a single node, the tokens are thus moved in each round. In order for them to remain detectable in this case, the status announced in $FLIP$ messages is `T` (whereas it is `N` for $SELECT$ messages).

**Regeneration:** The first thing a non-root node does after receiving the messages of the current round is to check whether the edge leading to its current parent is still present. If the edge disappeared, then the node regenerates a root

directly (Line 7). A nice property of the spanning forest principle is that this cannot happen twice in the same tree. And if a tree is broken into several pieces simultaneously, then each of the broken subtree will have exactly one node performing this operation.

**The unique score technique:** Unlike the high-level graph model from [10], in which the merging operation involved two nodes in an *exclusive* way, the non-atomic nature of message passing allows for a *chain* of selection that may involve an arbitrary long sequence of nodes (e.g. $a$ selects $b$, $b$ selects $c$, and so on). This has both advantages and drawbacks. On the good side, it makes the initial merging process very fast (see rounds 1 and 2 in Figure 2 to get an example). On the bad side, it is the reason why scores need to be introduced to avoid cycles. Indeed, relying only on a mere comparison of `id` to avoid cycles is not sufficient. Consider a chain of selection in round $i$ that ends up at some root node $u$. Nothing prevents $u$ to have passed the token to a lower-id child, say $v$, in the previous round $i-1$ (that same round when $u$'s status $T$ was overheard by the next-to-last root in the chain). Now, nothing again prevents $v$ to have selected one of the nodes in the selection chain in round $i$, thereby creating a cycle. The score mechanism prevents such a situation by enforcing that after each FLIP, the new root has a larger score than its predecessor (see Lines 9 and 13 in Algorithm 2). The score mechanism also guarantees that the current set of scores (network-wide) is always a permutation of the initial set of scores. Hence, scores are always unique. All of these elements are crucial ingredients in the proofs of correctness of Section 4.

**A note about convergence:** Each token performs a random walk in its underlying tree. Hence, unless some of the trees are bipartite, the configuration will eventually (and with high probability) stabilize into a single tree per connected component if the network stops changing. Although convergence is not the main focus here, we believe that pathetic scenarios where some trees are bipartite can easily be avoided, by making the tokens stop for a random additional round at the nodes (*lasy* walk). This way, the symmetry of bipartiteness is eventually broken *w.h.p.*

## 4 Proofs of correctness

This section summarizes the correctness analysis of our algorithm. (Note: complete proofs are given in Appendix.) We first define a handful of instrumental concepts that help minimize the number of properties to be proven. Then, as we start formulating lemmas and theorems, we adopt more precise notations regarding the state of the system. Precisely, we denote by $(i^-)u.varname$ (resp. $(i^+)u.varname$) the value of variable $varname$ at node $u$ before (resp. after) round $i$. Notice that for any node $u$, round $i$, and variable $varname$, we have $(i^+)u.varname = ((i+1)^-)u.varname$. We use whichever notation is the most convenient in the given context.

```
1  repeat
2  |  SEND(outMessage);
3  |  mailbox ← RECEIVE();  // Received messages, indexed by sender id
4  |  neighbors ← mailbox.keys();                // All the senders ids
5  |  children ← children ∩ neighbors
   |  // Regenerates a token if parent link is lost
6  |  if status=N ∧ parent ∉ neighbors then
7  |  |  BECOME_ROOT();
   |  // Checks if the outgoing FLIP or SELECT (if any) was successful
8  |  if outMessage.action ∈ {FLIP,SELECT} ∧ outMessage.target ∈
   |  neighbors then
9  |  |  ADOPT_PARENT(outMessage)
   |  // Processes the received messages
10 |  contender ← ⊥;
11 |  contenderScore ← 0;
12 |  forall message ∈ mailbox do
13 |  |  if message.target = id then
14 |  |  |  if message.action = FLIP then
15 |  |  |  |  BECOME_ROOT();
16 |  |  |  ADOPT_CHILD(message);     // called for both FLIP or SELECT
17 |  |  else
18 |  |  |  if message.status = T ∧ message.score > contenderScore then
19 |  |  |  |  contender ← message.id;
20 |  |  |  |  contenderScore ← message.score;
   |  // Prepares the message to be sent
21 |  outMessage ← ⊥
22 |  if status = T then
23 |  |  if contenderScore > score then
24 |  |  |  PREPARE_MESSAGE(SELECT, contender);
25 |  |  else
26 |  |  |  if children ≠ ∅ then
27 |  |  |  |  PREPARE_MESSAGE(FLIP, random(children));
28 |  if outMessage = ⊥ then
29 |  |  PREPARE_MESSAGE(HELLO, ⊥);
30 ;
```

**Algorithm 1:** Main Algorithm

```
1  procedure BECOME_ROOT
2  |  status ← T;
3  |  parent ← ⊥;

4  procedure ADOPT_PARENT(outMessage)
5  |  status ← N;
6  |  parent ← outMessage.target;
7  |  if outMessage.action = FLIP then
8  |  |  children ← children\parent;
9  |  |  score ← min(score, mailbox[parent].score);

10 procedure ADOPT_CHILD(message)
11 |  children.add(message.id);
12 |  if message.action = FLIP then
13 |  |  score ← max(score, message.score);

14 procedure PREPARE_MESSAGE(action, target)
15 |  switch action do
16 |  |  case SELECT
17 |  |  |  outMessage ← (id, N, SELECT, target, score);
18 |  |  case FLIP
19 |  |  |  outMessage ← (id, T, FLIP, target, score);
20 |  |  case HELLO
21 |  |  |  outMessage ← (id, status, ⊥, ⊥, score);
```

**Algorithm 2:** Functions called in Algorithm 1.

### 4.1 Helping definitions

These definitions are not specific to our algorithm, they are general graph concepts that make the rest easier.

**Definition 1 (Pseudotree and pseudoforest).** *A directed graph whose vertices have outdegree at most 1 is a* pseudoforest*. A vertex whose outdegree is 0 is called a* root*. The weakly connected components of a pseudoforest are called* pseudotrees*.*

**Lemma 1.** *A pseudotree has at most one root.*

*Proof.* By definition, a pseudotree $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is connected, thus $|E_\mathcal{T}| \geq |V_\mathcal{T}| - 1$. If $\mathcal{T}$ has several roots, then at least two nodes in $V_\mathcal{T}$ have no outgoing edge. Since the others have at most one, we must have $|E_\mathcal{T}| \leq |V_\mathcal{T}| - 2$, which is a contradiction. □

**Lemma 2.** *If a pseudotree $\mathcal{T}$ contains a root $r$, then it has no cycle.*

*Proof.* Let $V_1 \subset \mathcal{T}$ be the set of nodes at distance 1 from $V_0 = \{r\}$. Since $r$ has outdegree 0, there is an edge from each node in $V_1$ to $r$. Since $\mathcal{T}$ is a pseudotree, these nodes have no other outgoing edge than those ending up in $V_0$. The same argument can be applied inductively, all nodes at distance $i$ having no other outgoing edges than those ending up in $V_{i-1}$. □

**Definition 2 (Correct tree and correct forest).** *At the light of Lemma 1 and 2, we define a* correct tree *(or simply a* tree*) as a pseudotree in which a root can be found. We logically define a* correct forest *(or simply a* forest*) as a pseudoforest whose pseudotrees are all trees.*

Finally, because forests are considered in a spanning context, we say that a pseudoforest $\mathcal{F}$ is a correct forest *on graph* $G$ iff $\mathcal{F}$ is a correct forest *and* $\mathcal{F}$ is a subgraph of $G$. Note that defining correct trees as pseudotree in which a root can be found is very instrumental. When the moment arrives, this will allow us to prove our algorithm correct simply based on the presence of a root.

### 4.2 Consistency

**Forest consistency:** At the end of a round, each node must have the same local view of its tree as its tree neighbors:

**Definition 3 (forest consistency).** *The configuration $C_i$ is forest consistent if and only if for all nodes $u$, $(i^+)u.parent = v \Leftrightarrow u \in (i^+)v.children$.*

The proof of forest consistency is established by Theorem 1 (in Appendix), based on consistency of the initial configuration (Lemma 3) and an induction on the round number (Lemma 18). Forest consistency allows us to define the output of the algorithm after each round $i$ as the mere content of the `parent` variable.

**Graph consistency:** At the end of round $i$, the values of `parent` variable should be consistent with the underling graph $G_i$.

**Definition 4 (graph consistency).** *The configuration $C_i$ is graph consistent if and only if for all nodes $u$, $(i^+)u.parent = v \Rightarrow \{u, v\} \in E_i$.*

This property is established by Corollary 1 (Appendix). Graph consistency allows us to say that the output of the algorithm forms a pseudoforest on $G_i$.

**Definition 5 (Resulting forest).** *Given a round $i \geq 1$, occurring on graph $G_i$, the graph $\mathcal{F}_i = (V, E_{\mathcal{F}_i})$ such that $E_{\mathcal{F}_i} = \{(u, v) : \{u, v\} \in E_i, (i^+)u.parent = v\}$ is called the* pseudoforest *resulting from round $i$.*

**State consistency:** As explained in Section 3.1, the variables `parent` and `status` are somewhat redundant, since the possession of a token is synonymous with being a root. The equivalence between both variables after each round is established in Lemma 4. The main advantage of this equivalence is that it allows us to formulate and prove a large number of lemmas based on whichever is the most intuitive to manipulate for the considered property.

### 4.3 Correctness of the forest

In this section, we prove that the resulting forest is always correct (Definition 2). To achieve that goal, we first define a validity criterion at the node level, which recursively ensures the correctness of the tree this node belongs to thanks to Definition 2 (*i.e.* the existence of a root implies correctness).

**Definition 6.** *A node $u$ is said to be valid at the beginning of round $i$ if either $(i^-)u.status = T$ or $(i^-)u.parent$ is valid.*

The correctness of the whole forest can thus be established through showing that 1) it is initially correct (Lemma 3 and 2) if it is correct after round $i$, then it is correct after round $i + 1$ (Theorem 2). The latter is difficult to prove, and it involves a number of intermediate steps that corresponds to a case analysis based on every action a node can perform (sending of FLIP messages, SELECT messages, etc.).

We first prove that a node $u$ that sends a successful FLIP to $v$ in a round, is valid at the end of that round (lemma 23) because at the end of that round $v$ is a root. The proof relies on the fact that during a given round, a node cannot receive a FLIP and send a SELECT or a FLIP (lemma 20).

We then prove some necessary properties on the `score` variable at each node. For instance, a node changes its score at most once during a round (Lemma 25 and 26). Also, the set of all scores are a permutation of the node identifiers after each round (Lemma 27).

Then we prove that a node that sends a successful SELECT in a round $i$, is valid at the end of that round (Lemma 36). This part is the most technical and

is the one that proves that chains of selection can not create cycles thanks to properties on the score variables.

Finally, we prove that all roots at the beginning of a round are still valid at the end of the round (lemma 37). Therefore, if all nodes are valid at the beginning of round, then they are also valid at the end of the round (theorem 2). Since they are initially valid (Lemma 3), they are thus always valid, which completes the proofs.

## 5 Simulation on real world traces (Infocomm 2006)

We verified the applicability of our algorithm to real world situations. The algorithm was implemented in the JBotSim simulator [9] and tested upon the `Infocomm06` dataset. This dataset is a record of the possible interactions between people during the Infocomm'06 conference. The resulting graph has the following characteristics: the number of nodes is 78 and the average node degree is 1.3. It should also be noted that an edge can appear at any time but the fact that it is still present is thereafter only tested every 120 seconds; this means that the presence time of an edge is a multiple of 120 seconds. We assumed that 10 rounds can be performed per seconds (Figure 3), which seems a reasonable assumption. The results show the average number of trees per connected component, averaged over 100 runs.

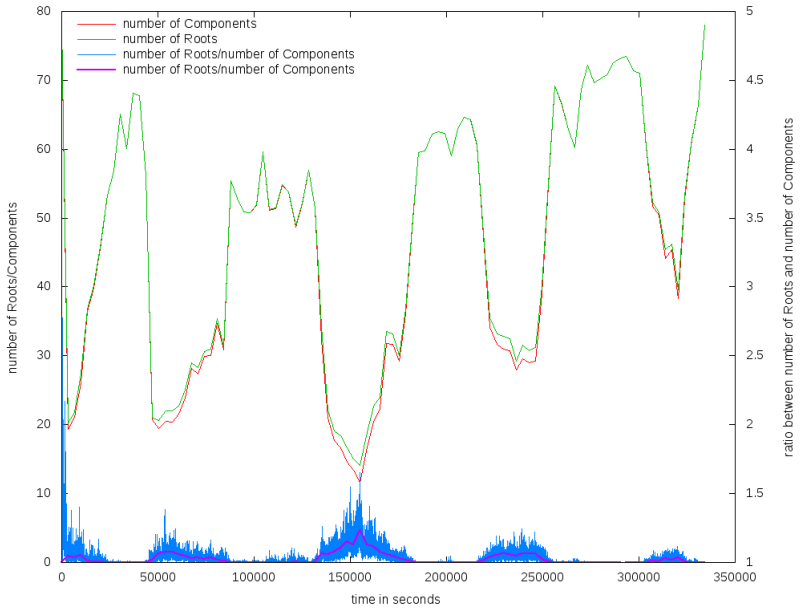



**Fig. 3.** Results of the simulation on Infocomm06 traces when assuming ten messages per second

These results show that the algorithm achieves an optimal configuration of a single spanning tree per connected component about 47% of the time, which is encouraging. It also demonstrates, incidentally, that the algorithm works correctly in such a real scenario.

## References


1. S. Abbas, M. Mosbah, and A. Zemmari. Distributed computation of a spanning tree in a dynamic graph by mobile agents. In *Proc. of IEEE Int. Conference on Engineering of Intelligent Systems (ICEIS)*, pages 1–6, 2006.
2. David Aldous and Jim Fill. Reversible markov chains and random walks on graphs, 2002.
3. Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of a spanning tree. *J. ACM*, 55(4):18:1–18:45, September 2008.
4. Baruch Awerbuch and Shimon Even. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 278–281. ACM, 1984.
5. H. Baala, O. Flauzac, J. Gaber, M. Bui, and T. El-Ghazawi. A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *Journal of Parallel and Distributed Computing*, 63:97–104, 2003.
6. Judit Bar-Ilan and Dror Zernik. Random leaders and random spanning trees. In Jean-Claude Bermond and Michel Raynal, editors, *Workshop on Distributed Algorithms (WDAG)*, volume 392 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 1989.
7. Thibault Bernard, Alain Bui, and Devan Sohier. Universal adaptive self-stabilizing traversal scheme: Random walk and reloading wave. *J. Parallel Distrib. Comput.*, 73(2):137–149, 2013.
8. Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In Andrzej Pelc, editor, *Distributed Computing*, volume 4731 of *Lecture Notes in Computer Science*, pages 92–107. Springer Berlin Heidelberg, 2007.
9. Arnaud Casteigts. The JBotSim library. *CoRR*, abs/1001.1435, 2013.
10. Arnaud Casteigts, Serge Chaumette, Frédéric Guinand, and Yoann Pigné. Distributed maintenance of anytime available spanning trees in dynamic networks. In *Proceedings of the 12th conference on Adhoc, Mobile, and Wireless Networks (ADHOC-NOW)*, volume 7960 of *Lecture Notes in Computer Science*, pages 99–110, Wroclaw, Poland, July 2013.
11. Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. Shortest, fastest, and foremost broadcast in dynamic networks. *CoRR*, abs/1210.3277, 2014.
12. Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 119–131. ACM, 1990.
13. Alex Kravchik and Shay Kutten. Time optimal synchronous self stabilizing spanning tree. In Yehuda Afek, editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 91–105. Springer Berlin Heidelberg, 2013.
14. Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM symposium on Theory of computing (STOC)*, pages 513–522. ACM, 2010.
15. Igor Litovsky, Yves Metivier, and Eric Sopena. Graph relabelling systems and distributed algorithms. In *Handbook of graph grammars and computing by graph transformation*. Citeseer, 2001.

# A Detailed proofs

## A.1 Consistency

**Lemma 3.** *The configuration $C_0$ is forest consistent and graph consistent. In $C_0$, the resulting pseudoforest is correct.*

*Proof.* The `parent` variable is initialized to $\perp$. So, the configuration $C_0$ is forest consistent and graph consistent. Any node $u$ belonging to the pseudotree $\mathcal{T}_u = (\{u\}, \emptyset)$. Each of these pseudotrees contains a root ($u$ itself) and is therefore a correct tree. □

We say that $u$ *sends a FLIP* (resp. *SELECT*) in round $i$ if and only if $(i^-)u.outMessage.action = FLIP$ (resp. *SELECT*). We say that it sends it *to node* $v$ if and only if $(i^-)u.outMessage.target = v$. Finally the FLIP or SELECT is said to be *successful* (resp. *failed*) if $\{u, v\} \in E_i$ (resp. $\{u, v\} \notin E_i$).

**Lemma 4 (state consistency).** *For all round $i \geq 0$, and for all node $u$, $(i^+)u.status = T \Leftrightarrow (i^+)u.parent = \perp$*

*Proof.* Initially, at any node $u$, $u.status = T$ and $u.parent = \perp$. The change of $u.status$ to $N$ always comes with the assignment of a non-null $identifier$ ($outMessage.target$) to $u.parent$ (procedure `ADOPT_PARENT()`), and assigning the value $T$ to $u.status$ is always followed by the change of $u.parent$ to $\perp$ (procedure `BECOME_ROOT()`). So at any configuration, $v.parent = \perp$ if and only if $v.status = T$. □

**Lemma 5.** *If $u$ does not send a FLIP or SELECT in round $i$, then $u$ does not execute the procedure `ADOPT_PARENT()` during round $i$.*

*Proof.* The execution of the procedure `ADOPT_PARENT()` by $u$ is conditioned by the sending of a SELECT or a FLIP by $u$ during the current round (line 8). □

**Observation 1.** At time where a node $u$ prepares its message to be sent during the round $i$, we have $u.parent = ((i-1)^+)u.parent$ (resp. $children, status$).

**Lemma 6.** *If $u$ sends a FLIP or SELECT in round $i$, then $(i^-)u.status = T$.*

*Proof.* $u$ sends in round $i$ the message prepared in round $i-1$. If $u$ sends a FLIP or a SELECT in round $i$ then in round $i-1$ `PREPARE_MESSAGE()` is called with FLIP or SELECT as action (lines 24 or 27). Both instructions are conditioned by $status = T$. □

**Lemma 7.** *If $v$ sends a message containing $T$ in round $i$, then $(i^-)v.status = T$.*

*Proof.* The procedure `PREPARE_MESSAGE()` is executed by a node $u$ in round $i-1$ to construct the message $m$ to be sent in round $i$. In all cases `PREPARE_MESSAGE()` sets $m.senderStatus$ to $T$ only if $u.status = T$. □

**Lemma 8.** *If $u$ sends a SELECT to $v$ in round $i$, then $(i^-)u.score < ((i-1)^-)v.score$.*

*Proof.* The value of the *score* field in the message sent by a node $v$ in round $i-1$ is $((i-1)^-)v.score$.

Assumes that the node $u$ sends a SELECT to $v$ in a round $i$. So, during the round $i-1$, $u$ sets its *contender* variable to $v$ and its *contenderScore* variable to $message.score$ *message* being the message sent by $v$ at the begining of round $i-1$. From that time to the end of round $i-1$, $u.score$ is not modified.

So $(i^-)u.score < ((i-1)^-)v.score$, if $u$ sends a SELECT to $v$ in a round $i$. □

**Lemma 9.** *If at the beginning of round $i$, the configuration is forest consistent then only $(i^-)u.parent$ can send a FLIP at destination of $u$ during the round $i$.*

*Proof.* A node $v$ can prepare a FLIP message to the node $u$ at then end of round $i-1$ only if $u \in (i^-)v.children$. We have $(i^-)u.parent = v$ according to the hypothesis (forest consistency at the beginning of round). Therefore, only the node $(i^-)u.parent$ can prepare a FLIP message at destination of $u$, at the end of round $i-1$. □

**Graph consistency:**

**Lemma 10.** *Let $u$ be a node such that $(i^-)u.parent \neq v \wedge (i^+)u.parent = v$. Then $u$ sends a successful FLIP or SELECT to $v$ during the round $i$.*

*Proof.* The only change of *parent* by $u$ to a non-null identifier $v$ in a round $i$ is at the execution of the procedure `ADOPT_PARENT()` which is conditioned by the reception of a message from $v$ (line 9). If $u$ receives the message of $v$ during round $i$ then $v$ effectively receives the message sent by $v$ (*reciprocal reception property*). □

**Lemma 11.** *Let $u$ be a node such that $(i^-)u.parent = v \wedge (i^+)u.parent = v$. We have $\{u, v\} \in E_i$.*

*Proof.* By Lemma 4, we have $(i^-)u.status = N$. So, $u$ does not send a FLIP or SELECT during the round $i$ (Lemma 6). Then, $u$ does not execute `ADOPT_PARENT()` during the round $i$ according to Lemma 5. Since $(i^+)u.parent = v$ we conclude that $u$ does not execute the procedure `BECOME_ROOT()` during the round $i$. So $u$ did receive a message from $(i^-)u.parent$ in round $i$. We have $\{u, v\} \in E_i$. □

**Corollary 1 (graph consistency).** *Every configuration is graph consistent.*

*Proof.* The configuration reached after any round is graph consistent (Lemmas 10 and 11). □

**Forest consistency:**

**Lemma 12.** *If* $(i^-)u.parent = v$ *then* $(i^+)u.parent = v$ *or* $(i^+)u.parent = \perp$.

*Proof.* According to Lemma 4, we have $(i^-)u.status = N$, so u cannot send a FLIP or a SELECT in round $i$ (by Lemma 6). Therefore, $u$ does not execute `ADOPT_PARENT()` in round $i$ (Lemma 5). We conclude that $(i^+)u.parent = v$ or $(i^+)u.parent = \perp$. □

**Lemma 13.** *Assume that at the beginning of round i, the configuration is forest consistent. If u receives a FLIP in round i, then it does not send a FLIP nor a SELECT in round i.*

*Proof.* We will establish the contraposition of the lemma statement: if $u$ sends a FLIP or a SELECT in round $i$, then it does not receive a FLIP in round $i$. By Lemma 6, we have $(i^-)u.status = T$. According to Lemma 4, $(i^-)u.parent = \perp$. Thus according to the hypothesis (forest consistency at the beginning of round), for any node $v$, $u \notin (i^-)v.children$. Therefore no node has prepared a FLIP message at destination of $u$, in round $i-1$. So $u$ cannot receive a FLIP in round $i$. □

**Lemma 14.** *Assume that at the beginning of round i, the configuration is forest consistent. If in round i, u changes u.parent to v then* $u \in (i^+)v.children : (i^-)u.parent \neq v \wedge (i^+)u.parent = v \Rightarrow u \in (i^+)v.children$.

*Proof.* $u$ sets $u.parent$ to $v$ only if the FLIP or SELECT was successful (Lemma 10). Therefore $v$ has received the FLIP or SELECT message sent by $u$.

The addition of a node $u$ to $v.children$ by $v$ is done during the excution of the procedure `ADOPT_CHILD()` which is conditioned by the reception of a FLIP or a SELECT message $m_u$ from $u$ ($m_u.target = v$, line 16). The procedure `ADOPT_CHILD()` is executed after line 5 which is the only instruction that could remove $u$ from $v.children$. So, $u \in (i^+)v.children$. We have $(i^-)u.parent \neq v \wedge (i^+)u.parent = v \Rightarrow u \in (i^+)v.children$. □

**Lemma 15.** *Assume that at the beginning of round i, the configuration is forest consistent. If in round i, v adds u to v.children then* $(i^+)u.parent = v : u \notin (i^-)v.children \wedge u \in (i^+)v.children \Rightarrow (i^+)u.parent = v$.

*Proof.* $v$ adds $u$ to $v.children$ only if it excutes the procedure `ADOPT_CHILD()` which is conditioned by the reception of a FLIP or a SELECT sent by $u$. As the reception of messages is reciprocal, $u$ also receives in round $i$ a message from $v$. This satisfies the condition for $u$ to execute the procedure `ADOPT_PARENT()` which sets $u.parent$ to $v$.

Only the execution of `BECOME_ROOT()` (at line 15) could modify the value of $u.parent$. This procedure would be executed only if $u$ has received a FLIP during round $i$ which cannot be the case. Notice that $u$ does not receive a FLIP during the round $i$ (Lemma 13). □

**Lemma 16.** *Assume that at the beginning of round $i$, the configuration is forest consistent. If in round $i$, $u$ changes $u.parent$ from $v$ to another value then $u \notin (i^+)v.children : (i^-)u.parent = v \wedge (i^+)u.parent \neq v \Rightarrow u \notin (i^+)v.children$.*

*Proof.* If $u$ changes $(i^+)u.parent$ then we have $(i^+)u.parent = \perp$ (Lemma 12). Only the execution of `BECOME_ROOT()` by $u$ sets $u.parent$ to $\perp$. The procedure `BECOME_ROOT()` is executed in two cases: at the detection of a disconnection (line 7), and at the reception of a FLIP message (line 15).

In the first case, the *reciprocal reception property* ensures that $v$ does not receive the message sent by $u$. So, $v$ removes $u$ from *children* (line 5).

In the second case, $u$ receives a FLIP from $(i^-)u.parent$ (Lemma 9). According to the *reciprocal reception property*, $v$ receives the message sent by $u$ during the round $i$. So, $v$ executes `ADOPT_PARENT(`$(i^-)v.outMessage$`)` which removes $u$ (i.e. $(i^-)v.outMessage.target$) from $v.children$ (line 9). □

**Lemma 17.** *Assume that at the beginning of round $i$, the configuration is forest consistent. If in round $i$, $v$ removes $u$ from $v.children$ then $(i^+)u.parent \neq v : u \in (i^-)v.children \wedge u \notin (i^+)v.children \Rightarrow (i^+)u.parent \neq v$.*

*Proof.* $v$ removes $u$ from $v.children$ in two cases: at the detection of a disconnection ($v$ does not receive a message from $u$, line 5), and when $v$ executes (`ADOPT_PARENT(`$(i).v.outMessage$`)`, line 9)

In the first case, the *reciprocal reception property* ensures that $u$ does not receive the message sent by $v$ during the round $i$. So, $u$ becomes a root : it executes the procedure `BECOME_ROOT()` (line 7).

In the second case, $v$ executes `ADOPT_PARENT(`$(i).v.outMessage$`)`. So $v$ did send a successful FLIP or SELECT (Lemma 5). As $v$ removes $u$ from $v.children$ during the execution of `ADOPT_PARENT(`$(i).v.outMessage$`)`, we have $(i^-).v.outMessage.target = u$ and $(i^-).v.outMessage.action = FLIP$ (see the procedure `ADOPT_PARENT(`$outMessage$`)`). So $v$ sends a successful FLIP to $u$ during round $i$. Therefore, in round $i$, $u$ executes the procedure `BECOME_ROOT()` (line 15): $u$ sets $u.parent$ to $\perp$. □

**Lemma 18 (Forest Consistency).** *Let $i$ be a round starting from a forest consistent configuration. The configuration reached at the end of round $i$ is forest consistent*

*Proof.* The configuration after the round $i$ is forest consistent according to Lemmas 14, 15, 16, 17. Notice that in the case where $u$ does not change the value of its parent variable (*resp.* $u$ stays in $v.children$) during round $i$, at the end of round $i$ the forest consistency property is preserved according to the contraposition of Lemma 17 (*resp.* contraposition of Lemma 16) and the hypothesis. □

**Theorem 1 (Consistency).** *Every configuration is forest consistent.*

*Proof.* $C_0$ is forest consistent (Lemma 3). The configuration reached after any round is forest consistent (Lemma 18). □

### A.2 Correctness of the forest

**Correctness of the resulting forest after token circulation:**

**Lemma 19.** *Let $v$ be a node. Only $(i^-)v.parent$ can send a FLIP at destination of $v$ during the round $i$.*

*Proof.* At the beginning of round $i$, the configuration is forest consistent (Theorem 1). Therefore, only the node $(i^-)v.parent$ can prepare a FLIP message at destination of $v$, at the end of round $i-1$ (Lemma 9). □

**Lemma 20.** *If $u$ receives a FLIP in round $i$, then it does not send a FLIP nor a SELECT in round $i$.*

*Proof.* At the beginning of round $i$, the configuration is forest consistent (Theorem 1). Therefore no node has prepared a FLIP message at destination of $u$, in round $i-1$ (Lemma 13). □

**Lemma 21 (Adoption).** *If $u$ sends a successful FLIP or SELECT to $v$ in round $i$, then $(i^+)u.status = N$ and $(i^+)u.parent = v$.*

*Proof.* In round $i$, $u.outMessage.action = FLIP$ or SELECT and $v \in (i^+)u.neighbors$. During the round $i$, $u$ executes the procedure `ADOPT_PARENT()` (line 9) which sets $(i^+)u.parent$ to $v$. According to Lemma 20, $u$ did not receive any FLIP message during the round $i$. Only an execution of `BECOME_ROOT()` by $u$ at line 15 can change the value of $u.parent$ during the round $i$. This line is not executed during round $i$. □

**Lemma 22.** *If $u$ sends a successful FLIP to $v$, then $(i^+)v.status = T$.*

*Proof.* $v$ received $mu$ in round $i$, so $\{u, v\} \in E_i$. $v$ executes the procedure `BECOME_ROOT()` that changes $v.status$ to $T$. After the execution of line 9, no instruction can set $v.status$ to $N$ until the end of round $i$. So $(i^+)v.status = T$. □

**Lemma 23.** *If $u$ sends a successful FLIP in round $i$, then $u$ is valid after round $i$.*

*Proof.* By Lemmas 21 and 22 $u$'s parent has a status $T$ after round $i$. □

**Proofs on score permutations:**

**Lemma 24.** *If $u$ sends a successful FLIP to $v$, then $(i^-)u.score \leq (i^+)v.score$.*

*Proof.* $u$ sent a message $mu$ to $v$ at the beginning of round $i$ such that $mu.action =$ FLIP, $mu.target = v.id$ and $mu.score = (i^-)u.score$. $v$ received $mu$ in round $i$, so $\{u, v\} \in E_i$. $v$ executes the procedure `ADOPT_CHILD(`$mu$`)` at line 16 in round $i$. This procedure sets the current score of $v$ to $max(v.score, mu.score)$, as $mu.score = (i^-)u.score$. After the execution of this instruction, we have $mu.score = (i^-)u.score \leq v.score$. We notice that after this operation, no instruction can change the value of $v.score$ (Lemma 19. □

**Lemma 25.** *$(i^-)u.score = (i^+)u.score$ unless $u$ sends or receives a successful FLIP in round $i$.*

*Proof.* $u$ changes its *score* value only by executing `ADOPT_PARENT(`$m_u$`)` or `ADOPT_CHILD(`$m_u$`)`. Both instructions that changes $u.score$ value in these procedures (Algorithm 2, line 9, 16) are conditioned by $m_u.action = FLIP$. □

**Lemma 26.** *A node $u$ changes $u.score$ at most once during a round.*

*Proof.* A node sends at most one FLIP message during a round. A node receives at most one FLIP message during a round (Lemma 19). Either a node receives a FLIP, sends one, or it does not receive and does not send a FLIP during a given round (Lemma 20). So, according to Lemma 25, a node changes $u.score$ at most once during a round. □

**Lemma 27.** *Before each round, the set of scores is a permutation of the set of identifiers.*

*Proof.* After the initialization in each node $u$, $u.score = u.id$. A node $u$ changes its score only by executing `ADOPT_PARENT()` or `ADOPT_CHILD()`. We will do a proof by induction. We assume at the beginning of round $i$, the set of scores is a a permutation of the set of indentifiers. We have for any node $u$, $mu.score = (i^-)u.score$.

According to Lemma 25, only a node sending or receiving a successful FLIP may change its *score* value. Assume that the node $u$ changes its *score* value during round $i$. Without lost of generality, we assume $u$ sends the successful FLIP to a node $v$ in round $i$.

By hypothesis, $u$ changes its *score* to $(i^-)v.score$ during the execution of `ADOPT_PARENT()` in round $i$. We have $(i^-)u.score\ geq(i^-)v.score$. $v$ executes the procedure `ADOPT_CHILD(`$mu$`)` at line 16 in round $i$. This procedure sets the current score of $v$ to $max(v.score, mu.score)$, as $mu.score = (i^-)u.score$. After the execution of this instruction, we have $v.score = (i^-)u.score$.

According lemma 26, we have $(i^+)v.score = (i^-)u.score$ and $(i^+)u.score = (i^-)v.score$. □

**Correctness of the resulting forest after mergings:** In lemmas 31 and 32, we establish that if $u$ sends a successful SELECT to $v$ in round $i$ either $(i^-)v.status = T$ or $(i^-)v.parent.status = T$. In the first case, we have $(i^-)u.score < (i^-)v.score$, and in the second case, we have $(i^-)u.score < (i^-)v.parent.score$. Let $ch$ be a series of nodes $u_0, u_1, u_2$ such that $(i^+)u_j.parent = u_{j+1}$ and such that $u_0$ sends a successful SELECT to $u_1$ during the round $i$. As a $ch$'s subchain of nodes having strictly increasing scores at the beginning of round $i$ may be built: $ch$ has not loop. So $ch$ ends by a node having a token : all nodes on that chain are valid.

**Lemma 28.** *If $v$ sends a message containing $T$ in round $i$ and $(i^+)v.status = N$, let $w = (i^+)v.parent$, then $(i^+)w.status = T$.*

*Proof.* If $v$ sends a message containing $T$ in round $i$, then $(i^+)v.status = T$. If $(i^+)v.status = N$, then $v$ has executed `ADOPT_PARENT()` in round $i$, because it is the only procedure that sets $v.status$ to $N$. $v$ executes `ADOPT_PARENT()` only if it has sent a FLIP message $m_v$ to a node $w$ ($m_v.action \neq$ SELECT because $m_v.senderStatus = T$), and if $w$ has received the message $m_v$ (*reciprocal reception property*). At the reception of $m_v$ by $w$, $w$ executes `BECOME_ROOT()` (line 16) which sets $w.status$ to $T$ and from this line until the end of the round no instruction can change $w.status$ to $N$. So $(i^+)w.status = T$.

At the execution of `ADOPT_PARENT()` by $v$, $v$ sets $v.parent$ to $w$. After this instruction there is only `BECOMES_ROOT()` that can modifie the value of $v.parent$, and which is conditioned by the reception of a FLIP message. According to lemma 20 $v$ cannot call `BECOMES_ROOT()` because it cannot receive a FLIP message. So $w = (i^+)v.parent$.

So, if $v$ sends a message containing $T$ in round $i$ and $(i^+)v.status = N$, and $w = (i^+)v.parent$, then $(i^+)w.status = T$. □

**Lemma 29.** *If $v$ sends a message containing $T$ in round $i$ and $(i^+)v.status = N$, let $w = (i^+)v.parent$, then $(i^+)w.score \geq (i^-)v.score$.*

*Proof.* We have $(i^-)v.status = T$ because in round $i-1$, $v.status$ cannot be modified after the execution of `PREPARE_MESSAGE()`. If $(i^-)v.children \neq \emptyset$ then $v$ sends a FLIP message to one of its children, named $u$, in round $i$. Either $\{u, v\} \in E_i$, then $(i^+)u.parent = v$, $(i^+)v.status = T$ and $(i^+)u.score \leq (i^-)v.score$ (see Lemmas 22 and 24). Otherwise $(i^+)v.status = T$. □

**Lemma 30.** *If $u$ sends a successful SELECT to $v$ in round $i$ then $((i-1)^-)v.status = T$.*

*Proof.* Node $u$ prepared a SELECT message to $v$ in round $i-1$, thus it had $u.contender = v$, which implies it received from $v$ a message containing $T$. We have then $((i-1)^-)v.status = T$ because after the execution of `PREPERE_MESSAGE()` by $v$ in round $i-2$, $v.status$ cannot be changed. □

**Lemma 31.** *If $u$ sends a successful SELECT to $v$ in round $i$ and $(i^-)v.status = T$, then $(i^-)u.score < (i^-)v.score$.*

*Proof.* By Lemma 30 $((i-1)^-)v.status = T$. Then Lemmas 8 and 25 respectively imply that $(i^-)u.score < ((i-1)^-)v.score$ and $((i-1)^-)v.score = (i^-)v.score$. □

**Lemma 32.** *If $u$ sends a successful SELECT to $v$ in round $i$ and $(i^-)v.status = N$, then let $w = (i^-)v.parent$. It holds that $(i^-)w.status = T$ and $(i^-)u.score < (i^-)w.score$.*

*Proof.* By Lemma 30 we have $((i-1)^-)v.status = T$. Then Lemmas 8 and 29 respectively imply that $(i^-)u.score < ((i-1)^-)v.score$ and $((i-1)^-)v.score \leq (i^-)w.score$. Lemma 28 implies that $(i^-)w.status = T$. □

**Lemma 33 (Cancellation).** *If $u$ sends a failed FLIP or SELECT in round $i$, then $(i^+)u.status = T$.*

*Proof.* By lemma 6, we have $(i^-)u.status = T$. $v$ did not receive the message from $u$ implies that $\{u, v\} \notin E_i$. So, in round $i$, $v \notin u.neighbors$ ($u$ did not receive the message from $v$). Only during the execution of `ADOPT_PARENT()`, called in line 9, $u$ can change its *status* to $N$. This procedure is not executed during the round $i$. □

**Lemma 34 (Conservation).** *If $(i^-)u.status = T$ and $u$ does not send a FLIP or SELECT in round $i$, then $(i^+)u.status = T$.*

*Proof.* By lemma 5, $u$ does not execute the procedure `ADOPT_PARENT()` during the round $i$. $u$ can set `status` variable to $N$ only if it executes `ADOPT_PARENT()`. □

**Lemma 35.** *If $(i^-)u.status = T$ and $u$ does not send a successful SELECT in round $i$, then $u$ is valid after the round $i$.*

*Proof.* According to Lemma 23, after the successful sending of a FLIP message in round $i$, $u$ is valid at the end of round $i$. If $u$ sends a failed SELECT or a failed FLIP then $u$ is valid after the round $i$ by Lemma 33. otherwise, $u$ did not send a SELECT or a FLIP during the round : it is also valid at the end of the round by Lemma 34. □

**Lemma 36.** *If a node sends a successful SELECT in round $i$, then it is valid at the end of round $i$.*

*Proof.* Let $S$ be the set of nodes that send a successful SELECT in round $i$ and are not valid at the end of round $i$. We will prove, by contradiction, that $S$ is empty. Assume $S$ is non-empty and consider the node in $S$ that had the largest score at the beginning of round (say, node $u$). Such a node exists by Lemma 27. We will prove that $u$ is valid after the round, which is a contradiction. Let $v$ be the recipient of $u$'s successful SELECT. By Lemma 21 $(i^+)u.parent = v$, thus is enough to show that $v$ is valid after round $i$ to get our contradiction. Let us examine both cases whether $(i^-)v.state = T$ or $N$.

If $(i^-)v.status = T$, then either $v$ also sends a successful SELECT in round $i$, or it does not. If it does not, then it is valid after round $i$ (Lemma 35). If it does, then it must be valid otherwise $u$ is not maximal in $S$ (Lemma 31).

If $(i^-)v.status = N$, then let $w = (i^-)v.parent$. Two cases are considered, whether $\{v, w\} \in E_i$ or not. If $\{v, w\} \notin E_i$ then $(i^+)v.status = T$ because the condition forces $u$ to call the procedure `BECOME_ROOT()` in line 7 which makes it take the status $T$. After, $u$ can takes the status $N$, only during the execution of the procedure `ADOPT_PARENT()` in line 9. This procedure is called by $u$ only if $u$ did send a FLIP or a SELECT at the beginning of round $i$ by lemma 5. By Lemma 6, this cannot happen. Thus $v$ is valid after round $i$. If $\{v, w\} \in E_i$, we use the fact that $(i^-)w.status = T$ (Lemma 28) to apply the same idea as we did above: either $w$ also sends a successful SELECT in round $i$, or it does not. If it does not, then it is valid after round $i$ (Lemma 35). If it does, then it must be valid otherwise $u$ is not maximal in $S$ (Lemma 32). □

**Correctness of resulting forest:**

**Lemma 37.** *If $(i^-)u.status = T$ then $u$ is valid after round $i$.*

*Proof.* According to Lemma 36, after the successful sending of a SELECT message in round $i$, $u$ is valid at the end of round $i$. According to Lemma 23, after the successful sending of a FLIP message in round $i$, $u$ is valid at the end of round $i$. If $u$ sends a failed SELECT or a failed FLIP then $u$ is valid after the round by Lemma 33. In otherwise, $u$ is also valid the round by Lemma 34.

□

**Theorem 2 (Resulting forest correctness).** *If all nodes are valid at the beginning of the the round $i$, then all nodes are valid after round $i$.*

*Proof.* Assume that a node $v$ is invalid after round $i$. According to Lemma 37, $(i^-)v.status = N$.

Let $u_0, u_1, u_2, ..., u_k$ be the finite series of nodes such that for $j \in [0, k-1]$, $(i^-)u_j.parent = u_{j+1}$, $(i^-)u_k.status = T$, and $u_0 = v$. This series exists because $u$ is valid at the beginning of round $i$.

Let $u'_1, u'_2, ...,$ be the infinite series of nodes such that for all $j \geq 1$ $(i^+)u'_j.parent = u_{j+1}$, and $(i^+)v.parent = u'_1$. This series exists because $v$ is invalid (by hypothesis).

According to Lemma 12, $j \in [1, k]$, $u_j = u'_j$. According to Lemma 37, $u_k$ is valid. So all nodes of the series $u_0, u_1, u_2, ..., u_k$ are valid. There is a contradiction.

□