RAIRO-Theor. Inf. Appl. 51 (2017) 51–70
DOI: 10.1051/ita/2017007

Available online at:
www.rairo-ita.org

MAINTENANCE OF A SPANNING TREE FOR DYNAMIC GRAPHS BY MOBILE AGENTS AND LOCAL COMPUTATIONS

Mouna Ktari[1], Mohamed Amine Haddar[2], Mohamed Mosbah[3] and Ahmed Hadj Kacem[4]

Abstract. The problem of constructing and maintaining a spanning tree in dynamic networks is important in distributed systems. Trees are essential structures in various communication protocols such as information broadcasting, routing, etc. In a distributed computing environment, the solution of this problem has many practical motivations. To make designing distributed algorithm easier, we model this latter with a local computation model. Based on the mobile agent paradigm, we present in this paper a distributed algorithm that maintain a hierarchical spanning tree in dynamic networks. We study all topological events that may affect the structure of the spanning tree: we address the appearance and the disappearance of places and communication channels.

Mathematics Subject Classification. 68.00.

1. Introduction

The continued evolution of distributed systems keeps the distributed computing an open area of research. Nowadays, distributed systems are dynamic and heterogeneous. The variation of components types and the absence of consensus in its modeling makes computation a very difficult task. To reduce modeling complexity we model both system and computations.

In order to perform a distributed computation in a dynamic context, we can model dynamic networks as a graph. Dynamic networks or dynamic distributed systems are characterized by a constant evolution in time. They are abstracted by dynamic graphs. The graph changes at any time through the addition or removal of nodes and/or edges. The best known model is the "random graph model" [1]. The network is modeled by a graph $G_{n,E}$ that denotes all graphs having n labeled nodes $V_1, V_2, ..., V_n$ and E edges. Each of these edges having the same probability to be chosen. A random graph $G'_{n,E}$ can be defined as an element of $G_{n,E}$ chosen at random. Several other models have been proposed, each one changes some of the properties of the initial

Keywords and phrases. Dynamic networks, distributed algorithms, mobile agents, local computation models, spanning tree.

[1] ReDCAD, University of Sfax, FSEGS 3018 Sfax, Tunis. mouna.ktari@redcad.org

[2] Information technology department, Taif University, Saoudi Arabia. haddar.amine@gmail.com

[3] LaBRI, CNRS, Bordeaux INP, University of Bordeaux, 33405 Talence, France. mohamed.mosbah@labri.fr

[4] ReDCAD, University of Sfax, FSEGS 3018 Sfax, Tunis. ahmed.hadjkacem@fsegs.rnu.tn

Article published by EDP Sciences

random graph model, such as the distribution degree [3], clustering coefficient, size of the largest connected component, diameter of the graph [2], etc.

Modeling dynamic networks based on this model requires a fully acquaintance of evolution of dynamic graph. The web graph is an example among the wide range of dynamic networks modeled by random graph model [4]. This model proposes creation of dynamic graphs with observed characteristics in the web graph and particularly that of mimicking its dynamics. As a consequence, the evolution of the dynamic graph is fully known (initially) from the beginning. Moreover, the random graph model is formalized in a particular application context and it is sometimes difficult to make it out of this context. It can be concluded that this generator of dynamic graphs can not match to the real-world networks.

In real cases, we do not have a precognition of the evolution of the graph since changes are unpredictable. In order to be closer of the real cases, we adopt another model that covers an extremely varied set of dynamic networks. This model is known as the evolving graph model. It is one of the first developed models characterized by a lack of prior knowledge of the evolution of dynamic graphs [5].

Modeling dynamic distributed system as a dynamic graph is a first step to reduce the modeling complexity. What remains, is the modeling of distributed computations. Among the proposed models in literature we present the local computation model. Graph relabeling systems based on local computations [6] can be considered as a tool which allows us to encode distributed algorithms in a formal and unified way. A graph relabeling system is based on a set of relabeling rules which are executed locally. These rules, closely related to mathematical and logic formulas, are able to derive the correctness of distributed algorithms. A real implementation of rewriting system was done in the ViSiDiA project [7]. This project aims essentially to simulate and visualize distributed algorithms encoded by rewriting rules.

In a distributed computing environment, the problem of the spanning tree has many practical motivations. It was studied on the static and dynamic context. The majority of proposed algorithms are based on message passing communication model. A performance evaluation using this model was achieved by [10]. The implementation of the local computation model using message passing is based on the following principle: a distributed algorithm must be run on any entity (node) of a computing system. The application of a computing step must be preceded by a synchronization, but this synchronization not only delays the execution of the algorithm but also uses a great amount of communication resources. Throughout the execution, all computation entities have committed to run the same algorithm, even if sometimes the rewriting rules are been only applied to a limited number of entities.

In order to solve the problems faced at the experimentation level, [10] proposed another implementation of rewriting systems using mobile agents [8]. As a case study, a solution of a spanning tree computation in anonymous networks has been proposed and proved [9]. This solution is based on benefits obviously inherited from the use of the mobile agent unlike classical model based on stationary process model. The proposed solution does not require active processes in each host. The resources, on a given host, are solicited when a mobile agent arrives on it. At the implementation phase, another important benefit comes from the omission of the synchronization needed to implement several algorithms using the classical models (message passing) in asynchronous networks. In fact, computations are encapsulated within mobile agents.

Towards mobile agents, the distributed computing community was presenting an increasing interest due to their considerable reduction of network load and their overcoming of the network latency [11]. In fact, the implementation of local computations using mobile agents in a static context has confirmed this interest [9]. In the same context (local computation), it remains a challenge to prove the mobile agent computational power in dynamic networks. Our work is a step in this direction.

This paper is organized as follows: we present in the second Section a related works survey. The third section presents the computation step of the spanning tree. In the fourth section, we present our algorithm for the maintenance of a spanning tree or a forest of spanning trees following the detection of topological events. In the fifth section, we present the proof and correctness of proposed algorithm. The last section concludes the paper and draws some future work perspectives.

2. Related works

According to literature, stationary process models are the most used computational models for dynamic distributed systems. A dynamic distributed system is considered as a set of interconnected computing entities communicating through established links. These systems are characterized by the appearance or disappearance of entities and/or links between them. They are modeled by a graph where the vertices (places) denote entities and the edges denote the communication links. The computational activities are done, in these models, by concurrent execution of the stationary sequential processes. A spectrum of mechanisms exists for interprocess communication. The message passing mechanism allows processes to communicate without sharing the same address, they communicate *via* messages. However, in the shared memory mechanism, processes communicate *via* exchanging information: reading and writing data to the shared region. In fact, in the distributed stetting, many distributed algorithms depend on the considered communication mechanisms. In general, a distributed algorithm which is designed and implemented in a given model becomes obsolete in another model. Even if it is possible, there is a need to re-adapt or to re-encode the algorithm depending on the model specification. In this context, graph relabeling systems based on local computations [6] can be viewed as a tool which allows to encode distributed algorithms in a formal and unified way. These model are characterized by their scalability. They are intensively studied and many results are then proposed in the static context [14–18] and in the dynamic context [19, 20], etc.

In [24], kuhn *et al.* proposed a distributed computation in dynamic networks. Authors consider a fixed set of nodes that operate in synchronized rounds and communicate by broadcast. In each round the communication graph is chosen adversarially, under an assumption of T-interval connectivity: throughout every block of T consecutive rounds there must exist a connected spanning sub-graph that remains stable. Based on proposed model, authors have shown how k-committee election can be used to solve counting and token dissemination.

Among possible computations in dynamic environment, we will focus on the spanning tree problem. In [25], the fully dynamic algorithm for maintaining a minimum spanning tree in time $o(\sqrt{n})$ per operation was studied. In the same context, an experimental comparison of several versions of dynamic trees: ST-trees, ET-trees, RC-trees, and two variants of top trees (self-adjusting and worst-case) was presented in [26]. In both previous cited works, authors have considered the problem of the tree during an arbitrary sequence of edge insertions and deletions.

Nowadays, dynamic environment are distributed. The spanning tree problem is among problems which has been very widely studied in dynamic distributed systems. The most of proposed algorithms are based on message passing communication model. In [27], authors provide a tree maintenance protocol with amortized communication complexity that is linear in the actual size A of the connected component in which the algorithm is performed. The message complexity of the proposed solution was not bounded as a function of A.

To achieve a distributed computation, the local computational model based on message passing communication model was well adopted. So, in the follow we will refer only to the proposed solutions who have adopted the local computations model. An original algorithm was proposed to building and maintaining a forest of spanning tree in highly dynamic networks (asynchronous case) [21]. This algorithm allows the maintenance of a non-minimum spanning forest in unrestricted dynamic networks, using an interaction model inspired from graph relabeling systems [22]. The synchronous case was studied by Barjon *et al.* in [19]. Authors are addressed to the maintenance of forest of spanning trees without any kind of assumption on the rate of changes. This algorithm guarantees the maintenance of a minimum spanning forest, but it is significantly more complex compared with a coarse-grain interaction algorithm proposed by Casteigts [21].

Relying on the local computation model, Haddar [10] has carried an evaluation of performance of distributed algorithms using the message passing communication model. The synchronization was identified as a main problem and indispensable in order to apply rules on non-disjoined sub-graphs. This synchronization not only delays the execution of the algorithm but also uses a great amount of communication resources. In order to solve the problems faced at the experimentation level, Haddar has proposed another model using mobile agents. The same paradigm has been used in the dynamic context by Baala *et al.* [20] and Abbas *et al.* [23]. Each agent

uses a random walk to move in the network and to construct a spanning sub-tree. With respect to dynamicity, both algorithms require the nodes to know an upper bound on the cover time of the random walk, in order to regenerate an agent if they are assumption requires the knowledge of the number of nodes or the network size. As a consequence, the efficiency of the timeout approach decreases dramatically with the rate of topological changes that may affect the structure of the graph and the network size. In particular, if topological changes are more frequent than the cover time, the computed tree is constantly fragmented into sub-graphs without a root.

All previously presented works have considered only the dynamic of communication links. Whereas in dynamic networks links can appear and disappear as well as places (nodes). In our work, we propose a distributed computation considering dynamic links and places using a mobile agent-dependent approach.

We are motivated then by the necessity of designing and proving distributed algorithms using mobile agent in dynamic networks. In [12], we have integrate the mobile agent on the conceptual level and we have proposed a computation model to describe, understand and prove dynamic distributed algorithms of mobile agents. Our model allows the use of properties defined in the mobile agent paradigm such as moving autonomously over the distributed system. When it visits a site, a mobile agent is also allowed to modify the site local information.

To illustrate our model simplicity, we propose in this paper, a solution to the spanning tree using mobile agent. A first variant of the maintenance algorithm of computed spanning tree has been shown in [13], where we proposed a solution when communication channels appears and disappears in the underling network. However, in this paper we present a second variant of the maintenance algorithm of computed spanning tree where places and links are dynamic.

3. Computation of a hierarchical spanning tree

Referring on proposed model in [12], we model dynamic distributed systems using evolving graphs $G = G_1, G_2, \ldots$ Every element G_i corresponds to a snapshot of the topology after a topological change occurs (we call it transaction). We suppose that the computation step of a hierarchical spanning tree starts and finishes on the initial connected graph G_1 before a topological event occurs and no software or hardware fault can occurs during the computation. Such assumptions can be considered real while we use a very quick algorithm to compute the spanning tree.

At time t_0, our system is composed by a collection $\mathbb{P}$ of places and one mobile agent and each place has a unique identity denoted by $place_{id}$. Initial placement of mobile agent is described by an injection $\pi_0 : \mathbb{A} \longrightarrow \mathbb{P}$. For all places composed the graph G_0 and for the mobile agent is associated to the initial state denoted by λ_0.

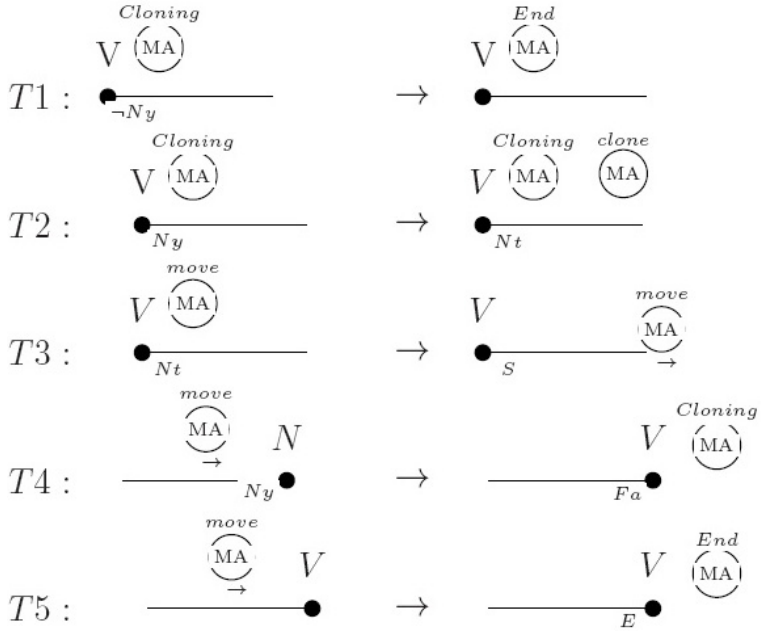
In our algorithm we need two variables: the $level_{id}$ and the $tree_{id}$. The aim of proposed algorithm is to create a *hierarchical spanning tree*. This means, for each place belonging to the spanning tree is associated a level. We denote this level by $level_{id}$ which is stored in every place local memory called the *whiteboard*. Moreover, each added place to the tree must know to which tree it belongs: this information is saved in a local variable called $tree_{id}$.

Let $(\mathcal{MobAS})$=$(\mathbb{A}, \mathbb{P}, \mathbb{S}, \pi_0, \lambda)$ a mobile agent system, $C_0 = (\texttt{state}_0, \mathbb{D}_0, \mathbb{A}_0)$ is a initial configuration such as:

- $\exists p \in \mathbb{P}$ such as p is the *home* et $\texttt{state}_0(p) = (Root, N_y, N_y, \ldots, N_y)$
- $\forall p \in \mathbb{P} \backslash home$, $\texttt{state}_0(p) = (N, N_y, N_y, \ldots, N_y)$
- $\mathbb{A} = a, \pi_0(a) = home$ et $\texttt{state}_0(a) = (\text{"}Cloning\text{"})$
- $\mathbb{D} = \emptyset$

At time t_0, the mobile agent is placed on the *Root*. This place, as all places composed the graph, has a unique identity $place_{id}$. The $level_{id}$ of this place will be initialized by 1, and the $tree_{id}$ will be initialized by the Root place's identity.

A mobile agent on the *Cloning* state, seeks ports labeled N_y. If it does not exist the mobile agent moves to the *End* state (see transition $T1$). But, if it exists, the mobile agent executes transition $T2$. It creates to every founded port a clone and changes the ports labels from N_y to N_t. Each clone, in the *move* state, takes the



With the following property: $T2 > T1$

FIGURE 1. Computation of the spanning tree.

$level_{id}$ and the $tree_{id}$ of the current place before leaving the place. It changes the output port label from N_t to S while leaving the place (see transition $T3$). Arriving to the destination place, execution of transition $T4$ or $T5$:

- If the state of visited place is N (no yet visited by another mobile agent), the mobile agent updates the local $level_{id}$ by that brought with it plus 1. After that, it changes the place state from N to V and the arrival port label from Ny to F_a. It updates the $tree_{id}$ of the current place with this brought from the last visited place. Finishing this task, it changes its state from *move* to *Cloning* (transition $T4$).
- If the state of visited place is V, the mobile agent changes the input port label to E and its state to *End* (transition $T5$).

Our algorithm is recursive, it is executed by a set of mobile agents operating in parallel in order to construct a *hierarchical spanning tree*. The spanning tree is composed by parents places and sons places related by channels labeled $S \longleftrightarrow Fa$. Adjacent place to the port S is a parent place: crossing a port S leads to the son place and crossing a port Fa leads to the parent place. Indeed, crossing ports Fa guide always, from any place, to the *Root* of the spanning tree. Proof and correction of computation step of a hierarchical spanning tree was clearly described in [12].

4. MAINTENANCE OF A HIERARCHICAL SPANNING TREE OR FOREST OF SPANNING TREES

The computed spanning tree can be affected following the detection of topological events: the appearance or disappearance of places leads to the occurrence and dissipate of communication channels (links). The disappearance of a place implies the removal of one or more communication channels which leads, in some cases, to the sub-division of the spanning tree into two sub-trees. Our algorithm must maintain a *hierarchical spanning tree* or a *forest of hierarchical spanning trees*.

The disappearance of a place causes the dissipate of one or more adjacent communication channels. Only places have lost the access path to the *Root* of the spanning tree (adjacent places to the port Fa) triggers the maintenance step. On the other hand, when a new place appear, it will join the spanning tree by connecting one or more communication links. The maintenance step is triggered by places where new channels are associated.

Before detailing the maintenance step, we assume that maintaining step triggered by topological events have to be done before the occurrence of the next topological event. During the computation step of the spanning tree, each place archives a copy of the mobile agent (in an inactive state). This copy will be activated upon the detection of the appearance or disappearance of communication channels. The activated copy of mobile agent updates the whiteboard of the current place. Thereafter, depending on the label of the adjacent port to the channel in question (added or removed one), the mobile agent must either do an update or turn back to the inactive state.

4.1. Topological event: Disappearance of a place

The disappearance of a communication channel can be the result of:

(1) the disappearance of a place from the spanning tree;
(2) or the displacement of a place in the underlying network.

Detecting the disappearance of a communication channel, each adjacent place activates a mobile agent (the disappearance of channels according to 1. or 2.). Each agent verifies the label of the adjacent port to the deleted channel. Three cases are possible:

Label E: the adjacent channel is *Excluded* from the spanning tree. The disappearance of this channel does not affect the structure of the spanning tree and activated mobile agent has nothing to do.

Label S: crossing this port, the copy agent can reach a son place. The loss of adjacent channel does not affect the access path from the current place to the *Root* of the spanning tree. Consequently, the activated mobile agent has nothing to do.

Label Fa: the loss of adjacent channel means losing the access path to the *Root* of the spanning tree. As a result, the current place (concerned place) and all derived sons places (sub-graph) are excluded from the spanning tree. However, we must try to find another path to re-associate again the concerned place or all the sub-graph to the spanning tree. In some cases (which are generally rare) it is possible that this path does not exist which causes the decomposition of the spanning tree in two sub-trees.

To re-associate again the concerned place (or all the sub-graph) to the hierarchical spanning tree, we must exploit an *Excluded* channel. This channel can be adjacent directly to the concerned place or adjacent to the derived sons places. The absence of the *Excluded* channels implies the sub-division of the spanning tree in two sub-trees. Finding an *Excluded* channel, the exploitation is possible only if the *Excluded* link **leads** to a place that has the ***level lower or equal to that of the concerned place***. This is a **very important rule of the maintenance step**.

In a case where *Excluded* channel is adjacent directly to the concerned place, the maintenance step is conducted in two phases: the *verification* and *exploitation* phase. However, if the *Excluded* channel is adjacent to the derived sons places, the maintenance step is conducted in three phases: the *verification*, *exploitation* and *cleaning* phase.

4.1.1. ***Excluded channel is adjacent directly to the concerned place***

Finding more adjacent excluded channels to the concerned place, the copy agent send thought each adjacent port E a get_{id} agent. Arriving to the destination place, each one takes with it the level of the visited place and turn back from the same crossed *Excluded* link. The copy agent select the adjacent channel leading to the place having the level lower or equal to that of the concerned place.

Case 1. *Excluded* channel is adjacent, on the other side, to the place having the ***level lower or equal*** to that of the concerned place. The copy agent creates an *Examiner* agent which will leave through the port labeled

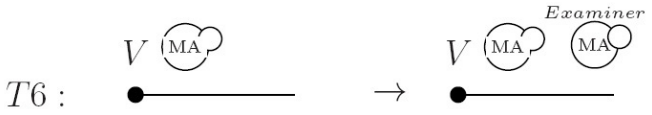


FIGURE 2. The copy agent creates an *Examiner*.

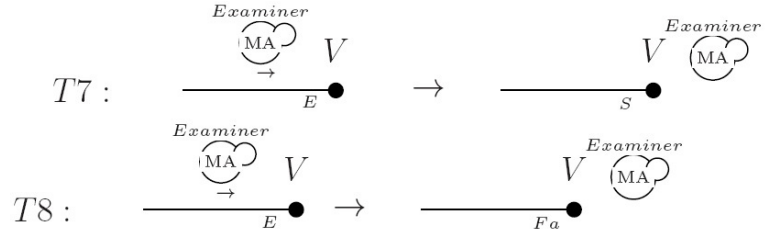
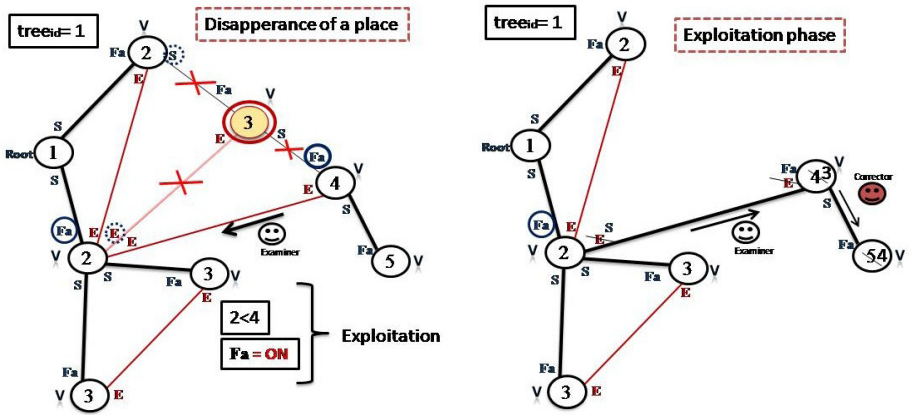


FIGURE 3. Exploitation of an Excluded channel by the *Examiner* agent.



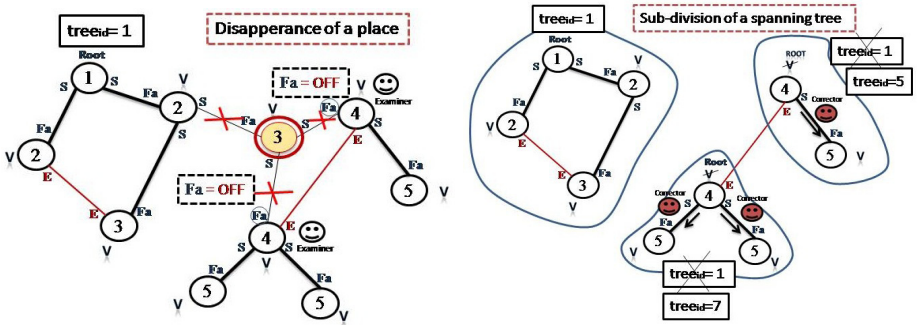
(a) Disappearance of a place (b) Spanning tree after the maintenance step

FIGURE 4. Maintenance of the spanning tree following the exploitation of a directed adjacent *Excluded* channel ($level_{id} <$).

E (see transition $T6$). Arriving to the destination place, before starting the *exploitation* phase, the *Examiner* must check the state of the port Fa adjacent to the visited place: set to ON or OFF.

- Status ON: *exploitation* phase (see Fig. 4): the *Examiner* changes the input port label from E to S, takes with it the $level_{id}$ of the current place and it leaves through the same arrival port (execution of transition $T7$). Arriving to the concerned place, the mobile agent changes the arrival label from E to Fa (execution of transition $T8$), computes the difference between the level of a reached place and that it took with it. Finding, the difference result different from 1, the *Examiner* agent affects the level brought with it plus 1 to the concerned place. This update requires updating of levels of all sons places (if they exists). So, this update will be ensured by *Corrector* agents.
  Operating principle of *Corrector* **agents**: the *Examiner* agent seeks ports labeled S. It creates to every founded port a *Corrector* agent and changes its state to *End*. Each *Corrector* agent takes the $level_{id}$ of the current place and crosses a port S. Arriving to the destination place, each one updates the local $level_{id}$

FIGURE 5. Transition executed by the *Examiner* agent when it is impossible to exploit an Excluded channel.



(a) Disappearance of a place (b) Forest of Spanning trees

FIGURE 6. Sub-division of the spanning tree on three sub-trees.

by the brought level with it plus 1. After that it seeks the adjacent ports S. If it finds, it should continue the updating of levels of derivatives sons places by creation of other *Corrector* agents. Indeed, the updating levels procedure is recursive. A *Corrector* agent changes its state to *End* following the creation of *Corrector* agents or where arriving on a sheet place.

- Status OFF: *verification* phase: that implies that the *Excluded* link could never re-associate again the dissociated place (or a sub-graph) to the spanning tree. The *Examiner* turns back. Arriving to the concerned place, it seeks another port E.
  - Finding another port E, the *Examiner* agent leaves through it. Arriving to the destination place, it must check the state of the port Fa adjacent to the visited place: set to ON or OFF and the same previous transitions are executed.
  - If it does not exist, the agent changes the place state from V to *Root* (see transition $T9$). The execution of this transition implies the sub-division of the hierarchical spanning tree to two sub-trees (see Fig. 6). Each one is characterized by its own $tree_{id}$ (see the definition of this variable on Sect. 3). The *Examiner* update the local $tree_{id}$ by identity of the current place ($place_{id}$). Such change is followed by a propagation of the new $tree_{id}$ towards all derived sons places. So, this propagation will be ensured by *Corrector* agents. The *Examiner* agent seeks ports S. It creates to every founded port a *Corrector* agent and changes its state to *End*. Each one takes the new $tree_{id}$ and crosses a port S. Arriving to the destination place, each agent affects the brought value of the local $tree_{id}$. It seeks again adjacent ports S. If it finds, it should continue the propagation of the new $tree_{id}$ towards derivatives sons places using *Corrector* agents. Indeed, the propagational procedure is recursive. A *Corrector* agent changes its state to *End* following the creation of *Corrector* agents or where arriving to a sheet place.

Case 2. *Excluded* channel is adjacent, on the other side, to the place having the ***level higher*** to that of the concerned place. The copy agent verifies if the *Excluded* link allows the mitigation to the place which verifies the rule of the maintenance step (see (a) in Fig. 8). It creates an *Examiner* agent which takes with it the $level_{id}$ of the current place before leaving through the port E (see transition $T6$). Arriving to the destination place it seeks the port labeled Fa and leaves through it. Coming to the parent place (through the port S), the

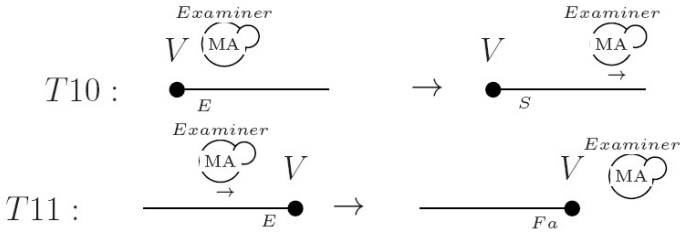
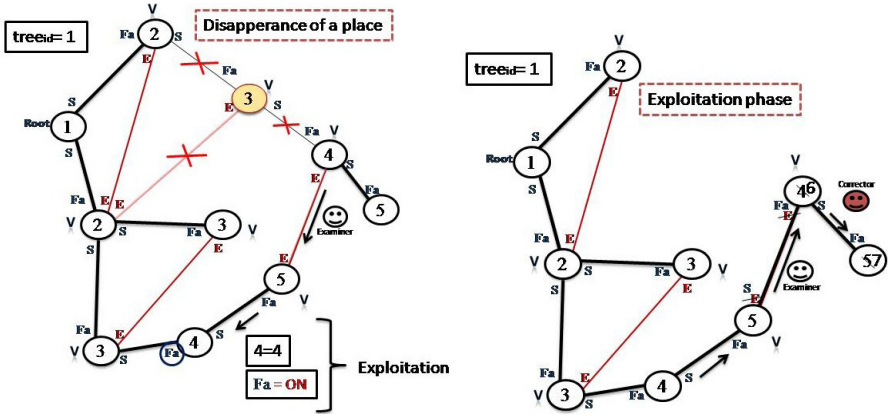


FIGURE 7. Exploitation of an Excluded channel by the *Examiner* agent.



(a) Disappearance of a place

(b) Spanning tree after the maintenance step

FIGURE 8. Maintenance of the spanning tree following the exploitation of a directed adjacent *Excluded* channel ($level_{id} >$).

Examiner marks this latter and compares brought level with that of the visited place. The crossing of Fa ports is iterative as the *Examiner* has not yet reached a place with the *lower or equal level*. Arriving to the desired place, it checks the state of the port labeled Fa. Two cases are possible: status ON or OFF:

- **Status** ON: *exploitation* phase (see (b) in Fig. 8): the *Examiner* turns back, following already marked ports S during the *verification* phase, until the attenuation of adjacent place to the *Excluded* channel. Tacking with it the level of the current place, the *Examiner* changes the output port label from E to S. Arriving to the destination place, it changes the arrival label from E to Fa (execution of transitions $T10$ and $T11$). Thereafter, the agent updates the local $level_{id}$ by that brought with it plus 1. *Corrector* agents have a responsibility to update levels of all sons places derived from the concerned place.
- **Status** OFF: that implies that the *Excluded* channel could never re-associate again the dissociated place (or a sub-graph) to the spanning tree. The *Examiner* turns back, following already marked ports S during the *verification* phase. Arriving to the concerned place, it seeks another port E. If it does not exist, the agent changes the place state from V to *Root*. The execution of this transition implies the sub-division of the hierarchical spanning tree to two sub-trees. Each one is characterized by its own $tree_{id}$. The *Examiner* update the local $tree_{id}$ by identity of the current place ($place_{id}$). Such change is followed by a propagation of the new $tree_{id}$ towards all derived sons places. So, this propagation will be ensured by *Corrector* agents (execution of transition $T9$).

FIGURE 9. Agent is located on the sheet place.

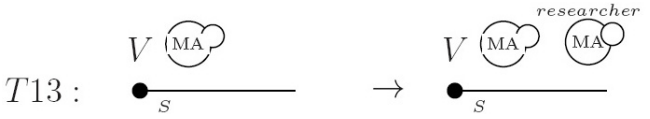


FIGURE 10. The copy agent creates a *researcher* agent.

4.1.2. ***Excluded channel is adjacent to the derived sons places***

In this case, no adjacent *Excluded* channels to the concerned place. The copy agent seeks ports S. If it does not exist, the mobile is located on the sheet place. It changes the state of the place from V to *Root* and turn back to the inactive state (the Standby mode): sub-division of the spanning tree on two sub-trees (see transition $T12$).

But, if it exists, the mobile agent creates to every founded port a *Researcher* agent (see transition $T13$). Each one takes the $level_{id}$ of the current place and starts **searching** of exploitable *Excluded* links. Finding an *Excluded* link, each agent is supposed to **verify** if this link can re-associate again the concerned place (or all a sub-graph) to the spanning tree.

To achieve our goal, we need a variable which is attached to each port. This one is denoted by $Nbr - Lab$: it is composed of the *number of port* and its *label*. Tow values are possible: $Nbr - Lab = Yes$ implies that the crossing of the adjacent channel leads towards the maintenance of the spanning tree, but $Nbr - Lab = No$ implies that the crossing of the adjacent channel leads to the sub-division of the spanning tree on two sub-trees. This variable is initialized by 0 value: no agent has passed through this port.

- **Verification phase**: after creation of *Researchers* agents, the copy agent goes from the *Active* mode to the *Standby* mode (inactive state), in which it sleeps on a place waiting for a wake up event: the first returned *Researcher* allows it to return to the *Active* mode. Each created agent leaves through a port S. Arriving to a son place, it searches an adjacent port E.
  Finding only one adjacent port labeled S, the same *Researcher* agent continues to seek *Excluded* links on the derived sons places. But, finding two or more adjacent ports labeled S, the *Researcher* creates to every founded port a *Researcher* agent and switch from *Active* to the *Standby* mode. It transmits to each created agent the level of the concerned place. A first returned *Researcher* agent allowing it to return to the *Active* mode. After its activation, a such *Researcher* verifies the type of returned response. If it is ***positive***, the activated agent sends immediately this response to its creator agent. Whereas, the activated agent waits the receipt of responses from other already sent agents. At the end, the activated agent must send a response to its creator agent: ***positive*** or ***negative*** response.
- **Exploitation phase**: receiving a ***positive*** response, the copy agent creates an *Examiner* agent. This latter, leaves through the port with $Nbr-S = Yes$. It crosses all ports S ($Nbr-S = Yes$) until reaches the adjacent place to the *Excluded* channel. Arriving to the desired place, and before crossing this link, the *Examiner* changes the output port label from E to Fa. Reaching the destination place, the *Examiner* modifies the input port label from E to S. Before turning back, the agent takes with it the $level_{id}$ of the current place. Going on a destination place, the *Examiner* computes the difference between the level of the current place and that brought with it. If the result is different to 1, the *Examiner* updates the local $level_{id}$ by that brought with it plus 1. This update requires updating levels of all sons places (if they exist). So, this update

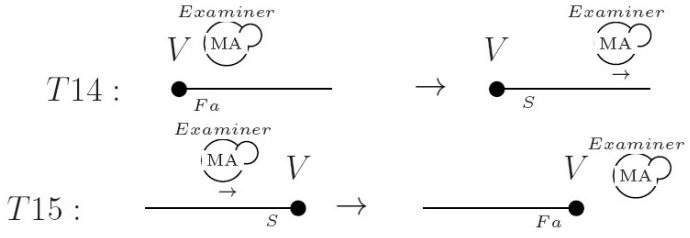


FIGURE 11. The *Examiner* reverses labels of crossed channels.

will be ensured by *Corrector* agents. Contrariwise, where the difference result is equal to 1, no update of the level of the current place, consequently no update of sons places levels.
Operating principle of *Corrector* agents was explained in the Case 1 under Sect. 4.1.1. After achieving the updating levels of sons places, the *Examiner* seeks the adjacent port labeled Fa. Before leaving, it takes with it the $level_{id}$ of the current place and changes the label of the output port from Fa to S. Arriving to the destination place, it modifies the label of the input port from S to Fa and affects taken level with it plus 1 to the visited place.
The reversing of labels of crossed channels (transitions $T14$ and $T15$) and the updating of labels of visited places (implicitly updating of levels of derived sons places) are repeated actions, while the *Examiner* agent has not yet reached the concerned place (that adjacent to the deleted link). Arriving to this place, the *Examiner* updates the local $level_{id}$ by that brought with it plus 1 (implicitly those of derived from). The agent finishes the maintenance step by ensuring the cleaning phase.

– **Cleaning phase**: the *Examiner*, in this phase of the maintenance step, must reset to zero all values of the variables $Nbr - Lab$ (those which were modified in the *verification* phase). It seeks the adjacent ports labeled S and creates a set of *Cleans* agents similar to the founded ports and changes its state to *End*. Before leaving, each *Clean* agent modifies the value of the $Nbr - Lab$ to 0. Arriving to the destination place, the *Clean* agent seeks ports having a $Nbr - Lab \neq 0$. It creates an equivalent number of *Clean* agents. Each agent, before leaving, reset to zero the $Nbr - Lab$. Reaching a destination place, if the input port $Nbr - Lab \neq 0$, each *Clean* modifies it to 0. It searches all adjacent port having $Nbr - Lab \neq 0$ and continues the **cleaning** phase. Reaching a place where no possibility of changing, the *Clean* agent goes to the *End* state.

4.2. Topological event: Appearance of a place

The appearance of a new communication channel can be the result of:

(1) the appearance of a new place;
(2) or the displacement of a place in the underlying network.

The appearance of a new place begets its association, through one or more communication channels, to the spanning tree. The maintenance step is triggered by places where new channels are associated (appeared channels according to 1. or 2.). The appeared new place doesn't yet have a copy of the mobile agent. Only those already belonging to the spanning tree can activate the copy of mobile agent.

Detecting the appearance of a new communication channel, each adjacent place activates a copy agent. Each one ensures the updating of a local whiteboard (see (a) in Fig. 14). The sate of each adjacent port to the added channel will be marked ON, however labels will be marked Ny. Each activated agent creates a *Solving* agent which will cross the new channel (transition $T16$). Before leaving, it takes with it the $level_{id}$ of the current place and the $tree_{id}$ value. Each *Solving* agent end by changing the output port label from Ny to S (transition $T17$). Arriving to the destination place, the *Solving* agent verifies the state of the visited place. Two cases are possible: state set to N or V.

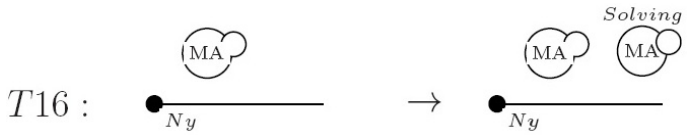
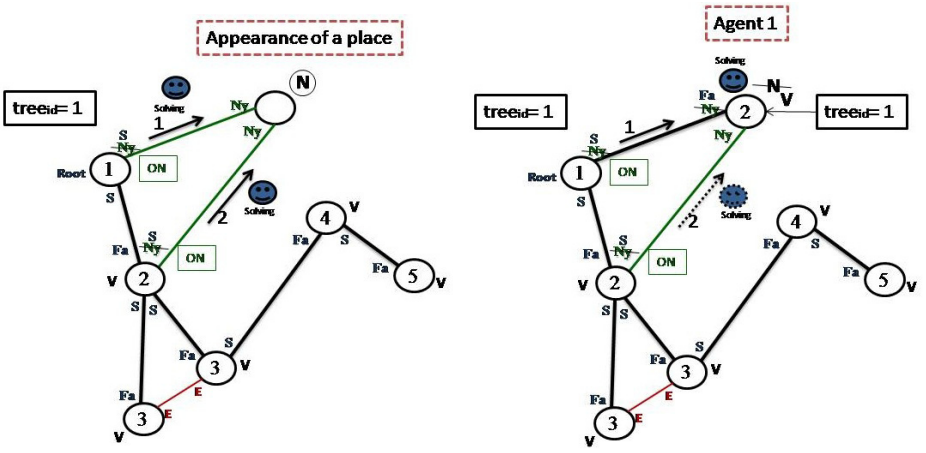


FIGURE 12. The copy agent creates a *Solving* agent.

FIGURE 13. Executed transition by the *Solving* agent before leaving a place.



(a) Appearance of a place

(b) Transitions executed by the first arrived *Solving* agent at the new place

FIGURE 14. Integration of the new place in the spanning tree.

– **State** N (Fig. 14 or Fig. 17): that means that the visited place does not belong to the spanning tree. It's a new place and it is no yet visited by another mobile agent. The *Solving* agent updates the local $level_{id}$ by that brought with it plus 1. After that, it changes the place state from N to V and the arrival port label from Ny to F_a. It updates the $tree_{id}$ of the current place with this brought from the last visited place. Finishing this task, it changes its state to *End*: the agent commits suicide.
– **State** V: that means that the visited place belongs to the spanning tree. The *Solving* agent verifies if a reached place and that from which it came belong to the same tree. It compares $tree_{id}$ variables.
  • ***The first case***: The equality between $tree_{id}$ implies that places belong to the same tree (see Fig. 15). The added communication channel must be *Excluded* from the spanning tree.
  Finding the label of the input port set to S, the *Solving* agent modifies the input port label to E and changes its state to *End* (transition $T18$). But finding it Ny (no yet crossed by another mobile agent), the *Solving* agent modifies the label of this port to E and turns back through the same arrival port. Arriving to the destination place it changes the label of reached port from S to E and its state to *End*.

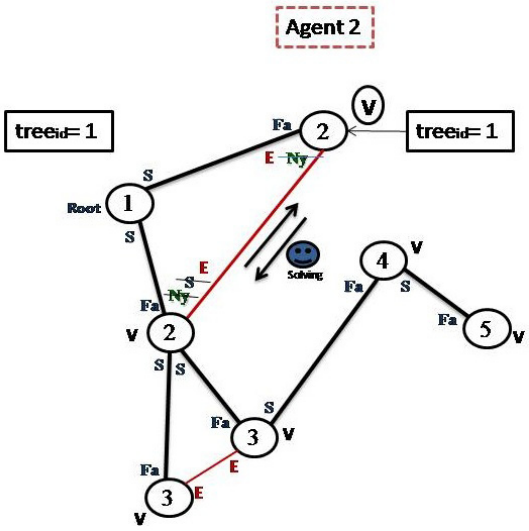


FIGURE 15. Transitions executed by the second arrived *Solving* agent at the added place.

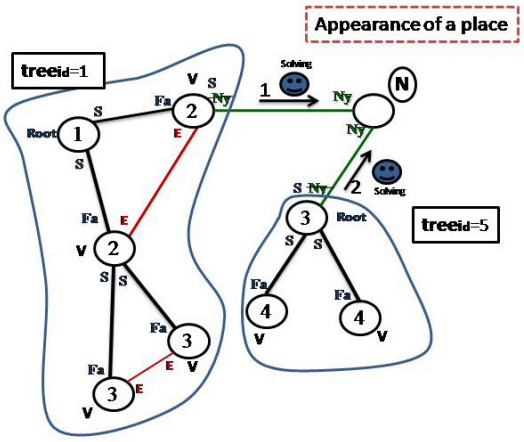
FIGURE 16. Executed transition by the *Solving* agent when it arrives to the place belonging to the same tree that the arrival place.

- ***The second case*** (see Fig. 18): Having two different $tree_{id}$ implies that two places, each one belongs to the tree. The added communication channel will be explored to merge two spanning trees on a single spanning tree. Before starting the merging action, the *Solving* agent verifies the label of the input port.

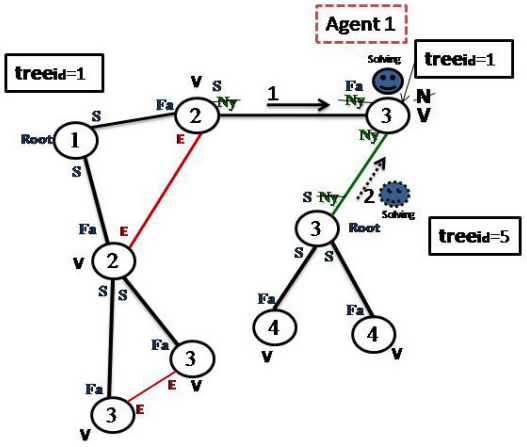
– **Finding it labeled Ny, this indicates only one mobile agent is currently working to merge spanning trees.**
$\Longrightarrow$ Having the lower $tree_{id}$, the *Solving* agent modifies the input label from Ny to Fa (see a little more down to follow up the merging process).
$\Longrightarrow$ Having the higher $tree_{id}$, the *Solving* agent edits this value by that of the current place. After that, it takes with it the $level_{id}$ of this place and turns back from the same arrival port (see (b) in Fig. 18). Before leaving, it modifies the label of the out put port from Ny to S (see a little more down to follow up the merging process).

– **Finding it labeled S, this indicates that two mobile agents are currently working to merge spanning trees.** But the merging can be performed by a single *Solving* agent. We suppose then, the *Solving* agent taking with it the higher $tree_{id}$ changes its state to *End*. The remaining agent modifies the input label from S to Fa.

Merging process: We will present in the following, the necessary details to accomplish the merging of two spanning trees on a single tree. The *Solving* agent starts then by verifying the state of the current place. Finding it V, no change. But, finding it *Root*, the *Solving* agent must modify it from *Root* to V (see (b) in Fig. 18). Following the updating of the state, the *Solving* agent must edit the $tree_{id}$ by that brought with it. It followed after that by updating the level of the current place by that brought with it plus 1. The *Solving* agent seeks



(a) Appearance of a new place



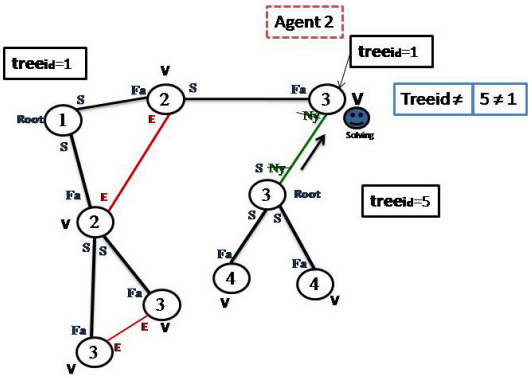
(b) Transitions executed by the first arrived *Solving* agent at the new place

FIGURE 17. Integration of the new place in the spanning tree (second case).

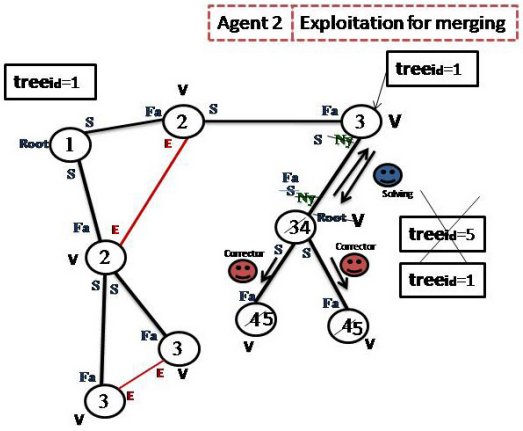
sons places, if they exist, it updates their levels and propagate toward them the new $tree_{id}$ using *Corrector* agents. These agents have the same operating principal of those created by the *Examiner* agent.

Once the *Solving* agent has launched *Corrector* agents, it checks if the current place has a second port Fa, two cases are possible:

- The absence of the second port Fa implies that the *Solving* agent is on the *Root* of the second spanning tree, it changes its state to *End*.
- The presence of the second port Fa implies that the *Solving* agent is on a place belonging to the second spanning tree. This later has already a *Root*. In order to eliminate the instance of two roots, the *Solving* agent must reaching this root place. Before leaving through the second port Fa, the *Solving* agent takes with it the $level_{id}$ of the current place and the $tree_{id}$. It modifies the label of the output port from Fa to S. Arriving to the destination place, it changes the label of the input port from S to Fa, affects the level brought with it plus 1 to the visited place and edits the $tree_{id}$. Having a sons places, the *Solving* agent must ensuring the updating of their levels and propagates toward them the new $tree_{id}$ using *Corrector* agents.



(a) Transitions executed by the second arrived *Solving* agent at the new place



(b) Exploitation for merging

FIGURE 18. Merging of two spanning trees following the appearance of a new place.

Indeed, the reversing labels of crossed channels (transitions $T19$ and $T20$), the updating of the visited places levels and the propagation of the $tree_{id}$ are the repeated actions while the *Solving* agent has not yet reached the *Root* place. Arriving to this place, the *Solving* agent modifies the state of the place from *Root* to V. It updates the level of this place and those of derived sons places and concatenates by updating the $tree_{id}$ and its propagation towards derived sons places (using *Corrector* agents) and finish by changing its state to *End*.

5. Proof and correction of maintenance of a hierarchical spanning tree

5.1. Topological event: Disappearance of a place

Lemma 5.1. *Following the disappearance of a place belonging to the spanning tree one or more communication channels are dissipate. The maintenance step is triggered by places that are adjacent to the dissociated channels.*

Proof. Detecting the disappearance of a communication channel, each place adjacent to the dissociated channel activates the mobile agent. Each one verifies the label of the adjacent port to the deleted channel: port E

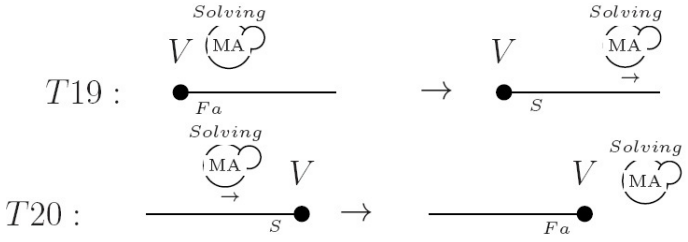


FIGURE 19. The *Solving* agent reverses labels of crossed channels.

(*Excluded* link from the tree), port S (leads to the son place), and port Fa (leads to the parent place). See the explanation below. □

Lemma 5.2. *The disappearance of the Excluded communication channel from the spanning tree does not affect the structure of the tree.*

Proof. Detecting removal of a communication channel, the adjacent execution platform activates a copy agent which will ensure the updating of a local whiteboard. The sate of the adjacent port of the deleted channel changes from ON to OFF state, however labels remain unchanged. Verifying the label of the adjacent port and finding it labeled E, that implies that the adjacent link is *Excluded* from the spanning tree and its deletion does not affect the structure of the computed spanning tree. □

Lemma 5.3. *The disappearance of a communication channel belonging to the spanning tree causes the spacing of adjacent place to the port Fa and all derived sons places (sub-graph) from the spanning tree.*

Proof. The spanning tree is composed from sons places and parents places related through channels labeled $S \longrightarrow Fa$. From a port labeled S, the agent can reach a son place. The loss of the adjacent channel does not affect the access path to the *root*. But, the loss of the adjacent channel of the port labeled Fa means losing of the access path to the parent place and implicitly the access path to the *Root* of the spanning tree. In this case, the adjacent place to the port labeled Fa and all derived sons places will be dissociated from the spanning tree. □

Lemma 5.4. *Following the sub-division of the hierarchical spanning tree to two sub-trees, each one must have its own Root.*

Proof. Two cases are possible:

- Located on the sheet place (no adjacent ports other Fa port), the mobile agent modifies the state of the current place from V to *Root*. It achieves the necessary updates and changes its state to *End*.
- Arriving on a place through an *Excluded* channel and finding the state of the adjacent port Fa set to OFF, that implies that the *Excluded* link could never re-associate again the dissociated place to the spanning tree. The mobile agent turns back until arriving to the concerned place (adjacent place to the deleted link). It modifies the state of the place from V to *Root*, it achieves the necessary updates and changes its state to *End*. □

Lemma 5.5. *Following the sub-division of the hierarchical spanning tree to two sub-trees, each one must have its own $tree_{id}$.*

Proof. The sub-division of the hierarchical spanning tree causes the generation of two sub-trees. According to Lemma 5.4, each one have its own *Root*. After updating the state of the place from V to *Root*, the mobile agent modifies the local $tree_{id}$ by the identity of the current place ($place_{id}$). Such change is followed by a propagation

of the new $tree_{id}$ to all sons places derived from the root place. The agent seeks ports labeled S. It creates a set of $SetTree_{id}$ agents similar to founded ports and changes its state to *End*. Operating principle of $SetTree_{id}$ agents was explained in the Case 2 under the sub section. □

Lemma 5.6. *Following the exploitation of an Excluded channel, levels of some places must be updating.*

Proof. An *Excluded* channel is adjacent to two places. On one side, it can be adjacent to the concerned place or to the son place. On the other side, it can be adjacent to the place having a *lower or equal* level or to the place having a *higher* level (compared by that of the concerned place). Be placed on the place compatible with the second side, the mobile agent takes with it the level of this place. It turns back by browsing the *Excluded* channel. Arriving to the destination place (that's compatible with the first side), it checks the necessity of updating the level of the visited place. If it's necessary, the *Examiner* affects the level brought with it plus 1 to the visited place. It seeks ports labeled S and creates a set of *Corrector* agents similar to the founded ports. Operating principle of *Corrector* agents was explained in the Case 1 under Sect. 4.1.1.

After achieving the updating levels of sons places, the *Examiner* seeks the adjacent port Fa. It leaves through it until reaching the concerned place. During the travel, it updates levels of visited places (implicitly the updating of sons places). Arriving to the concerned place, the *Examiner* updates the level of this one and that of sons places (if they exist). □

5.2. Topological event: Appearance of a place

Lemma 5.7. *The appearance of a new place begets its association, through one or more communication channels, to the spanning tree. The maintenance step is triggered by places where new channels are associated.*

Proof. Detecting the appearance of a new communication channel, the adjacent execution platform activates a copy agent (only places belonging to the spanning tree have a copy of mobile agent). Each one creates a *Solving* agent. Before crossing the new link, each one takes with it the local $tree_{id}$ and the $level_{id}$. Arriving to the destination place, the *Solving* agent verifies the state of the visited place. Two cases are possible: N or V.

- state set to N: the visited place does not belong to the spanning tree, its a new place;
- state set to V: the visited place belong to the spanning tree. The *Solving* agent verifies if a reached place and that from which it came belonging to the same tree. It compares $tree_{id}$ variables and according to the result, it reacts. □

Lemma 5.8. *The appearance of a new communication channel between two places belonging to the same tree implies the Exclution of this link from the spanning tree. The structure of the spanning tree remains unchanged.*

Proof. Arriving to the destination place, the *Solving* agent compares a brought $tree_{id}$ with it and that of the visited place. If $tree_{id}$ are equals the crossed link must be *Excluded* from the spanning tree. □

Lemma 5.9. *In merging case of two sub-trees, the new spanning tree must have a unique Root place.*

Proof. In merging case, only one *Solving* agent has a responsibility to achieve the fusion of spanning trees. It verifies the state of the current place. Finding it set to *Root*, the *Solving* agent modifies the sate from *Root* to V. But finding it set to V, the *Solving* seeks the adjacent port Fa. It leaves through it until reaching the *root* place. Arriving on this place it modifies the sate from *Root* to V. □

Lemma 5.10. *In merging case of two sub-trees, all places composing the new spanning tree must have the same* $tree_{id}$.

Proof. In merging case, the *Solving* agent edits the $tree_{id}$ of the current place by that brought with it. It seeks ports labeled S. It creates a set of $SetTree_{id}$ agents similar to the founded ports. These agents are loaded to propagate the $tree_{id}$ for all derived sons places. After achieving the propagation of the new $tree_{id}$ to the sons places, the *Solving* seeks the adjacent port Fa. If it find, it leaves through it until reaching the *root* place. During the travel, it propagates the new $tree_{id}$ to the visited places (implicitly the propagation to the sons places). Arriving to the *root* place, the *Solving* edits the $tree_{id}$ of this one and that of sons places (if they exist). □

Lemma 5.11. *In merging case of two sub-trees, levels of some places must be updating.*

Proof. In merging case, carries already with it the level of the place from which it came, the *Solving* agent checks the necessity of updating the level of the visited place. If it's necessary, it affects the level brought with it plus 1 to the local variable $level_{id}$. It seeks ports labeled S and creates a set of *Corrector* agents similar to the founded ports. These agents are loaded to update levels of derived sons places. After achieving updating levels of sons places, the *Solving* seeks the adjacent port Fa. If it finds, it leaves through it until reaching the *root* place. During the travel, it updates levels of visited places (implicitly the updating of sons places). Arriving to the *root* place, the *Solving* updates the level of this one and that of sons places (if they exist). □

Lemma 5.12. *Just before the next topological event occurs (appearance or disappearance of places and/or communication channels), any spanning tree or sub-tree has its own Root.*

Proof. Evident according to Lemmas 5.4 and 5.9. □

Lemma 5.13. *Just before the next topological event occurs, all places belonging to the same sub-tree have the same* $tree_{id}$.

Proof. Evident according to Lemmas 5.5 and 5.10. □

Lemma 5.14. *Just before the next topological event occurs, any sub-tree is a* ***hierarchical spanning tree****.*

Proof. Evident according to Lemmas 5.6 and 5.11. □

Theorem 5.15. *Just before the next topological event occurs, any sub-graph induced by channels having one port labeled* Fa *is a spanning tree.*

Proof. Evident according to Lemmas 5.12–5.14. □

Theorem 5.16. *A necessary time to complete the maintenance step is on* $O(m)$.

Proof. A necessary time to complete the maintenance step is the time of travel of a longest path belonging to the spacing sub-tree (e.q added sub-tree) following the detection of the deletion of a communication channel (e.q addition). We suppose that this sub-graph is composed from m places, whence the necessary time is on $O(m)$. □

6. Conclusion and future works

The construction and the maintenance of a spanning tree is one of the most important problems in a dynamic distributed system. Based on our proposed framework for designing, proving and simulating distributed algorithms in dynamic networks [12], we have presented in this paper a solution for the spanning tree problem. In our model we have integrated the mobile agent on the conceptual level. The mobile agent has a responsibility of both computing and maintaining spanning trees.

A first variant of the maintenance algorithm of computed hierarchical spanning tree has been shown in [13], where we proposed a solution when communication channels appear and disappear in the underling network.

A second variant of proposed algorithm (the maintenance of a spanning tree or a forest of spanning trees) is presented in this paper. We have studied all topological events that may affect the structure of the graph: we address the appearance and the disappearance of places and communication channels.

The disappearance of a place implies the removal of one or more communication channels. As far as the appearance of a place, this event can lead to the appearance of one or more communication channels. The displacement of a place in the underlying network can lead also to the appearance and the disappearance of communication links.

Detecting the disappearance of a communication channel belonging to the spanning tree, adjacent place to the port Fa will be discarded from the spanning tree. To re-associate again this place (or all a sub-graph) to the hierarchical spanning tree, we must exploit an *Excluded* channel. The appearance of a new communication channel, between two places each one belongs to the tree, will be explored to merge two spanning trees on a single spanning tree.

To validate our approach, we are proceeding to implement and experiment our algorithm using ViSiDiA project (*Visualization and Simulation of Distributed Algorithms*). We are planning in the near future to present the implementation results of proposed algorithm and presenting other solutions related to the distributed algorithms problems in the dynamic graphs (election, counting, coloring, etc.).

References

[1] A. Renyi and P. Erdos, On the evolution of random graphs, In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences* (1960) 17–61.

[2] B. Yener and C.C. Bilgin, Dynamic network evolution: Models, clustering, anomaly detection. IEEE Networks (2006).

[3] F. Chung, L. Lu and W. Aeillo, A Random Graph Model for Massive Graphs, In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing, ser.* Portland, Oregon, USA (2000) 171–180.

[4] A.L. Barabasi and R. Albert, Emergence of Scaling in Random Networks. Science **286** (1999) 509–512.

[5] A. Ferreria, On models and algorithms for dynamic communication networks: The case for evolving graphs, In 4^e *rencontres francophones sur les Aspects Algorithmiques des Telecommunications (ALGOTEL), Mèze, France* (2002).

[6] E. Sopena, I. Litovsky and Y. Métivier, Handbook of graph grammars and computing by graph transformation. World Scientific Publishing Co., Inc (1999).

[7] A. Sellami, M. Bauderon, M. Mosbah, S. Gruner and Y. Métivier, Visualization of Distributed Algorithms Based on Graph Relabelling Systems. *Electron. Notes Theoret. Comput. Sci.* **50** (2001) 227–237.

[8] A. Hadj Kacem, M.A. Haddar, M. Mosbah, M. Jmail and Y. Métivier, A Distributed Computational Model for Mobile Agents. In Agent Computing and Multi-Agent Systems, 10th Pacific Rim International Conference on Multi-Agents, PRIMA (2007) 416–421.

[9] A. Hadj Kacem, M.A. Haddar, M. Mosbah and Y. Métivier, Proving Distributed Algorithms for Mobile Agents: Examples of Spanning Tree Computation in Anonymous Networks. In *Distributed Computing and Networking, 9th International Conference, ICDCN* (2008) 286–291.

[10] M.A. Haddar, *Codage d'algorithmes distribués d'agents mobiles à l'aide de calculs locaux*. Ph.D. thesis. University of Bordeaux 1, France (2011).

[11] D.B. Lange and M. Oshima, Seven good reasons for mobile agents. Commun. ACM (1999) 88–89.

[12] A. Hadj Kacem, M. Ktari, M.A. Haddar and M. Mosbah, Proving Distributed Algorithms for Mobile Agents: Examples of Spanning Tree Computation in Dynamic Networks. In *12th ACS/IEEE International Conference on Computer Systems and Applications AICCSA* (2015).

[13] A. Hadj Kacem, M. Ktari, M.A. Haddar and M. Mosbah, Distributed Computation and Maintenance of a Spanning Tree in Dynamic Netrworks by Mobile Agents. In *The 30th IEEE International Conference on Advanced Information Networking and Applications AINA (2016).*

[14] I. Litovsky and Y. Métivier, Computing trees with graph rewriting systems with priorities. Tree Automata and Languages (1992) 115–140.

[15] M. Yamashita and T. Kameda, Computing on anonymous networks: Part I-characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems* (1996).

[16] M. Yamashita and T. Kameda, Computing on Anonymous Networks: Part II Decision and Membership Problems. In *IEEE Trans. Parallel Distrib. Syst. IEEE Press* (1996).

[17] I. Litovsky and Y. Métivier, Computing with graph rewriting systems with priorities. Theoretical Computer Science (1993).

[18] A. Muscholl, E. Godard and Y. Métivier, The power of local computations in graphs with initial knowledge. In Theory and applications of graph transformations, Vol. 1764 of *Lecture notes in computer science* (2000).

[19] A. Casteigts, C. Johnen, M. Bargon, S. Chaumette and Y.M. Neggaz, Maintaining a spanning forest in highly dynamic networks: The synchronous case. In vol. 8878 of *Lecture Notes in Computer Science*. Springer (2014)
[20] H. Baala, J. Gaber, M. Bui, O. Flauzac and T. El-Ghazawi, A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *J. Parallel Distrib. Comput.* **63** (2003) 97–104.
[21] A. Casteigts, F. Guinand, S. Chaumette and Y. Pigne, Distributed maintenance of anytime available spanning trees in dynamic networks. In ADHOC-NOW. In vol. 7960 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg (2013).
[22] E. Sopena and I. Litovsky, Graph relabelling systems and distributed algorithms. In *Handbook of graph grammars and computing by graph transformation*. World scientific (2001) 1–56.
[23] A. Zemmari, M. Mosbah and S. Abbas, Distributed Computation of a Spanning Tree in a Dynamic Graph by Mobile Agents. In *IEEE International Conference on Engineering of Intelligent Systems ICEIS* (2006).
[24] F. Kuhn, N.A. Lynch and R. Oshman, Distributed computation in dynamic networks. In *Proc. of the 42nd ACM Symposium on Theory of Computing, STOC* (2010) 513–522.
[25] M.R. Henzinger and V. King, Maintaining minimum spanning trees in dynamic graphs. In *Automata, Languages and Programming: 24th International Colloquium* (1997) 594–604.
[26] R.E. Tarjan and R.F. Werneck, Dynamic Trees in Practice. In *Experimental Algorithms: 6th International Workshop* (2007) 80–93.
[27] S. Kutten and A. Porat, Maintenance of a Spanning Tree in Dynamic Networks. In *Proc. of the 13th International Symposium on Distributed Computing* (1999) 342–355.

Communicated by S. Tison.
Received May 13, 2016. Accepted July 10, 2017.