HAL
open science

Distributed Maintenance of Anytime Available Spanning Trees in Dynamic Networks

Arnaud Casteigts, Serge Chaumette, Frédéric Guinand, Yoann Pigné

To cite this version:

Arnaud Casteigts, Serge Chaumette, Frédéric Guinand, Yoann Pigné. Distributed Maintenance of Anytime Available Spanning Trees in Dynamic Networks. Distributed Maintenance of Anytime Available Spanning Trees in Dynamic Networks, Jul 2013, Poland. pp.99-110. hal-00376701v2

HAL Id: hal-00376701

https://hal.science/hal-00376701v2

Submitted on 22 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed Maintenance of Anytime Available Spanning Trees in Dynamic Networks

Arnaud Casteigts[1], Serge Chaumette[1], Frédéric Guinand[2] and Yoann Pigné[2]

[1]LaBRI, University of Bordeaux, France
{arnaud.casteigts,serge.chaumette}@labri.fr

[2]LITIS, University of Le Havre, France
{frederic.guinand,yoann.pigne}@univ-lehavre.fr

Abstract. We address the problem of building and maintaining a forest of spanning trees in highly dynamic networks, in which topological events can occur at any time and any rate, and no stable periods can be assumed. In these harsh environments, we strive to preserve some properties such as cycle-freeness or existence of a unique root in each fragment regardless of the events, so as to keep these fragments functioning uninterruptedly to a possible extent. Our algorithm operates at a coarse-grain level, using atomic pairwise interactions akin to population protocol or graph relabeling systems. The algorithm relies on a perpetual alternation of *topology-induced splittings* and *computation-induced mergings* of a forest of trees. Each tree in the forest hosts exactly one token (also called root) that performs a random walk *inside* the tree, switching parent-child relationships as it crosses edges. When two tokens are located on both sides of a same edge, their trees are merged upon this edge and one token disappears. Whenever an edge that belongs to a tree disappears, its child endpoint regenerates a new token instantly. The main features of this approach is that both *merging* and *splitting* are purely localized phenomenons. This paper presents the algorithm and establishes its correctness in arbitrary dynamic networks. We also discuss aspects related to the implementation of this general principle in fine-grain models, as well as embryonic elements of analysis. The characterization of the algorithm performance is left open, both analytically and experimentally.

1 Introduction

Spanning trees are essential components in communication networks. The availability of such structures simplifies a large number of tasks, among which broadcasting, routing, or termination detection. From the standpoint of distributed computing, constructing a spanning tree implies the collaboration of neighboring nodes in order to establish selective relationships that inter-connect the whole network without cycle.

The problem is very different in essence in static and dynamic networks. In a static network, there is generally a distinction between the construction of a tree

and its effective use, both taking place at different times. In truly dynamic networks (e.g. vehicular networks), the set of communication links evolves rapidly and continuously. As a result, the trees need to be updated on a constant basis and while they are used. Early works addressing the spanning tree problem in dynamic graphs (see e.g. [4, 10, 6] and the references therein) applied strong restrictions on the dynamicity; namely, these works assumed the network stabilizes eventually, or recurrently offers stable periods during which the tree can entirely be recomputed. These assumptions are certainly appropriate in the case of occasional failures or reconfigurations of the topology. But they are not reasonable in highly dynamic scenarios like mobile ad hoc networks.

We are interested in understanding what can still be done in the harshest dynamic context. In particular, we consider networks in which no stability period is ever expected; no information is available about future topological events; no restrictions apply to the rate of these events; and no contemporaneous end-to-end connectivity is assumed (that is, we address *delay-tolerant networks* [8]). On the other hand, we allow ourselves to reason at a high level of abstraction, using a coarse-grain interaction model akin to recent *population protocol* models [3]. While we find the problem in this model interesting in its own right, we still hope and believe the principles highlighted here can help subsequent effort to make it work in finer-grain (e.g. message passing) models.

The algorithm relies on a perpetual alternation of *topology-induced splittings* and *computation-induced mergings* of a forest of spanning trees. Each tree in the forest hosts exactly one token (also called root) that performs a random walk *inside* the tree, switching parent-child relationships as it crosses edges. When two tokens are located on both sides of a same edge, their trees are merged upon this edge and one token disappears. Whenever an edge that belongs to a tree disappears, its child endpoint regenerates a new token instantly. The main features of this approach is that both *merging* and *splitting* are purely localized phenomenons.

After reviewing some relevant work in Section 2, we define the network model and assumptions, as well as the computational model in Section 3. The algorithm is then presented in detail and proved correct in Section 4. This presentation is followed by a discussion regarding some important implementation choices (e.g. priority between different rules of interaction). In Section 5, we provide preliminary results on the analysis of the algorithm, which we regard as a coalescing particle system involving random walks in trees. We conclude in Section 6 with some perspectives.

2 Related Work

The problem of building distributed spanning trees in communication networks, and more generally in graphs, has been extensively studied during the last three decades and a large literature exists on the topic. It is noteworthy that the problem was studied by different communities (self-stabilization, stochastic processes, distributed computing) using different paradigms and terminologies (*e.g.*

token, mobile agent, random walk, legal state, stabilization time, coalescing time, tree, forest, etc.). We review below the most relevant concepts and approaches to solve this problem.

Self-stabilization: A system that reaches a *legal* state starting from an *arbitrary* state is called *self-stabilizing*. After a fault in the system, the time required to reach the legal state is called the *stabilization time*. In the context of spanning trees in dynamic networks, topological changes are the faults, and having the entire network covered by a single tree, or in case of partitioned networks one tree per connected component, is the legal state. One approach to transform a non-self-stabilizing algorithm into a self-stabilizing one, is to *reset* the states of the nodes when a fault occurs, so that a new execution of the algorithm is initiated. This approach has been considered by most self-stabilizing algorithms proposed so far for the spanning tree problem, and an optimal-time solution was proposed in [4] (as a coarse-grain graph algorithm, more recently transposed into the message passing model in [6]). We refer the reader to [10] for a more general survey on self-stabilizing spanning tree algorithms. In these works, the algorithms assume that no additional fault occur during the stabilization period, which is not acceptable in highly dynamic networks.

Random walk: A random walk is a sequence of nodes such that each node in the sequence (except the starting node) is randomly selected among the neighbors of its predecessor. Random walks have been used to solve several problems in distributed systems, such as leader election, voting, or spanning trees [7]. The idea of using random walks to compute spanning trees was first proposed by Aldous in [2], where a single random walk is considered. Anytime, the set of all covered nodes, along with the edges from which they were visited the first time, defines a random tree that spans the nodes already visited.

Mobile agents: Mobile agents are entities that can travel across the network, and perform tasks on the underlying nodes. These agents may or may not carry their own memory, and adopt a variety of strategies to move within the network. In [5], distributed random walks of mobile agents (called *tokens* in the paper) are used. More precisely, colored tokens are annexing territories while walking within the network. Each token builds a tree (a subtree of the global spanning tree). When two tokens meet or when a token visits a node that have already been visited, the two trees are merged into one. This operation is performed by a *wave propagation*, which is a broadcast-based process that occurs along the edges of the trees. The network is assumed connected and no topological changes are allowed during the construction of the tree. Unique identifiers are also required. A related approach was proposed in [1], where mobile colored agents (equivalent to tokens) construct subtrees that are progressively merged into a final spanning tree. Whenever one agent enters the region of another, the agent that have the larger color progressively takes control of the nodes and eventually destroys the other agent. The advantage of this gradual process is that it avoids the wave propagation. However, unique identifiers are still required to generate the colors

and some global information (an upper bound in the cover time of the random walk) is needed to regenerate an agent. Finally, the approach does not tolerate frequent topological events.

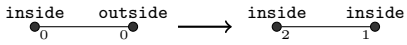
In comparison to these approaches, the one we propose does not require stable periods or unique identifiers (nor any global information). This is, to the best of our knowledge, the first attempt in this direction.

3 Network model and assumptions

We represent the network as an evolving graph $\mathcal{G} = \{G_1, G_2, ...\}$, all elements of which correspond to snapshots of the topology, and the transitions between them bijectively reflect the occurrence of one, or several simultaneous topological events (appearance or disappearance of edges). More elaborate variants of evolving graphs can be found in the original paper [9]. However, this basic variant is suitable enough for our purpose.

At a given moment, the network is therefore represented by an undirected simple graph $G_i = (V, E_i)$, where the set of nodes V is assumed to be constant, while the set of edges varies without restriction from one G_i to the next. The temporal span of each G_i is arbitrary and in particular, it is not bounded (whether from above or below). We do not require the existence of unique identifiers for the nodes, but we assume they are able to distinguish between their incident edges and assign a local value to them (thus, an edge typically has two values, one on each side). Note that in practice, especially in a wireless network, this feature would require unique identifiers to be implemented. It is however a weaker assumption from a theoretical standpoint. Further, it is more natural to think of our algorithm without identifiers.

3.1 Computational model

We consider a coarse-grain interaction model akin to *population protocols* [3] or *graph relabeling systems* [11]. In these models a computation step is an atomic pairwise interaction. Precisely, a computation step takes as input the state of a pair of nodes (together with their common edge), and modifies these states according to some rule. For example, the rule 

may represent the construction of a rooted spanning tree in a static network from some distinguished `inside` node. We assume in general that two interactions can occur in parallel so long as they are disjoint (they do not imply a common node). The way interactions are selected, that is, the *scheduling*, is typically not a part of the algorithm (e.g. it can be adversarial with some constraints, or probabilistic, or result from some finer-grain interaction). The general properties we establish on our algorithm are insensitive to these concerns. Note that the guard of a rule (left part) may represent two nodes in a same state. In this case, despite the absence of unique identifiers, symmetry is broken by the application of the rule – however, the choice of what role is played by each node is not controlled by the algorithm (it is up to the scheduler).

Dealing with a dynamic graph (the usual population protocols deal with static graphs), we consider another type of operation in addition to pairwise interaction. This operation, triggered by topological events, consists in updating the state of a node immediately after one of its edges disappears. As such, an algorithm can associate reactive operations to the loss of a link.

4 The spanning forest algorithm

Informally, the algorithm is based on three operations on tokens: *circulation*, *merging*, and *regeneration*, which aim at maintaining exactly one token per tree. Initially, every node forms a tree of its own and is the root of that tree (it has the token). When two token owners interact over a common edge, their tokens are merged into one and their common edge is added to the tree (*merging* rule r_1, see Figure 1 below). The parent-child relation is set accordingly. The rest of the time, each token performs a random walk along the edges of its own tree (*circulation* rule r_2, see Figure 2 below) in search of new merging opportunities; parent-child relations are flipped as the circulation proceeds, so that a node can always tell, locally, which edge leads to the token. Whenever an edge of the tree disappears, the node on the child side regenerates a token (*regeneration* rule r_a, see Figure 3 below), which re-enables its orphan tree to keep running the process.

4.1 State space and initialization

At any time, the state of the system is fully described by two functions: one function for the state of the nodes $\lambda : V \to \{T, N\}$, where T means this node has a token, while N means it does not; and one function for the state of the edges *locally to both endpoints* $\lambda : V \times E_i \to \{0, 1, 2\}$, where E_i is the current set of edges. The domain of both functions being different and non-ambiguous from the context, we authorize a unique symbol λ to denote them. State 0 for an edge means it does not belong to a tree. States 1 or 2 mean it does, and the local direction is from child to parent (state 1) or from parent to child (state 2). Hence, an edge whose state is 1 at one end, must be in state 2 at the other end. Notice that one bit of information is enough to encode the state of a node, and two bits, locally at each node, are sufficient for an edge.

Initialization: Given the first graph $G_0 = (V, E_0)$, we set $\lambda(v) = T$ for all $v \in V$. We also set $\lambda(v, e) = 0$ and $\lambda(u, e) = 0$ for all $e = (u, v) \in E_0$. In words, every node initially holds a token and none of the edges belong to a tree.

4.2 State transitions

The evolution of the process is determined by two sources of events: topological events (i.e., appearance or disappearance of an edge) and computational events (i.e., pairwise interaction). We specify both separately. Keep in mind the principle presented here is intended to be extremely general, and several important questions, like priority among rules or the role played by each node in the rule, are deliberately set aside at this point. (They are discussed shortly after.)

Transitions induced by pairwise interaction

Merging rule: Given two nodes u and v involved in an interaction over an edge $e = (u, v)$, the operation is specified as follows. If $\lambda(u) = T$ and $\lambda(v) = T$, then set $\lambda(v) = N$, $\lambda(v, e) = 1$, and $\lambda(u, e) = 2$. This rule, called *merging* rule (r_1), can be represented graphically as shown in Figure 1.

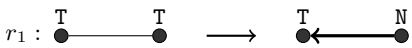


Fig. 1. Merging rule (graphical representation).

Circulation rule: Given two nodes u and v involved in an interaction over an edge $e = (u, v)$, the operation is specified as follows. If $\lambda(u) = T$ and $\lambda(v) = N$ and $\lambda(u, e) = 2$, then set $\lambda(u) = N$, $\lambda(v) = T$, $\lambda(v, e) = 2$, and $\lambda(u, e) = 1$. This *circulation* rule (r_2) can be represented graphically as shown on Figure 2.

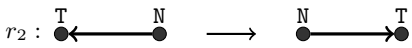


Fig. 2. Circulation rule (graphical representation).

Transitions induced by topological events

Given two consecutive graphs G_i and G_{i+1} in $\mathcal{G}$, the transition from one to the other induces the following updates on the states of the system.

Appearance of an edge: For all $e = (u, v) \in E_{i+1} \backslash E_i$, both $\lambda(u, e)$ and $\lambda(v, e)$ are set to 0. In words, new edges are initialized with state 0 on both sides.

Disappearance of an edge: For all $e = (u, v) \in E_i \backslash E_{i+1}$, if $\lambda(u, e) = 1$, then set $\lambda(u) = T$; else if $\lambda(v, e) = 1$, then set $\lambda(v) = T$. In words, if a node loses the edge leading to its parent, it regenerates a token immediately. This rule, called *regeneration* rule (r_a), can be represented graphically as shown on Figure 3.

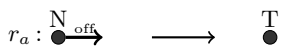


Fig. 3. Regeneration rule (graphical representation).

An example execution sequence of the algorithm is provided on Figure 4.

4.3 Correctness

In this Section we establish some properties of the spanning forest algorithm, namely, that there is always exactly one root (token) in every tree, and no cycle can possibly occur.

Lemma 1. *At any time, there is at least one token per tree.*

Proof. The lemma holds initially, when every node is the root of its own tree. Now observe that both *merging* and *circulation* operations perserve this property. Indeed, the application of r_1 merges two trees but suppresses one token, while r_2 just moves a token within the underlying tree. We can thus focus on the disappearance of edges. Whenever an edge e disappears, either e did not belong to a tree or it did. If it did not, nothing has to be done. If it did, then this tree is now split into two trees, one of which is left token-less. By rule r_a, whose application is immediate, a token is regenerated on the orphan side of that edge (edge state 1). If several such edges had disappeared simultaneously, the same mechanism would have occurred relative to each fragment. □

Lemma 2. *At any time, there is at most one token per tree.*

Proof (By contradiction). The only rule leading to the creation of a token is r_a. Since the lemma holds initially, the presence of more than one token in a tree must result from one of these events:

1. Rule r_a was applied despite the existence of another token in the tree.
2. Rule r_a was applied several times simultaneously in the tree.

In the first case, the contradiction stems from the fact that r_a is applied on the child endpoint of a lost edge. By construction, the token is thus on the other side and the local subtree is token free. In the second case, the contradiction is slightly less direct. Let v and v' be two nodes of a same tree, both of which have applied r_a simultaneously. Three cases are possible regarding the relative position of v and v' in the tree:

1. (a) v is an ancestor of v'. This is impossible because the application of r_a by v' results from the disappearance of its parent edge.
   (b) v' is an ancestor of v. Same argument for v.
   (c) v and v' have a common ancestor. This is again impossible because the application of r_a results from the disappearance of a parent edge, therefore neither v nor v' can have an ancestor at all. □

Theorem 1. *At any time, there is exactly one token per tree.*

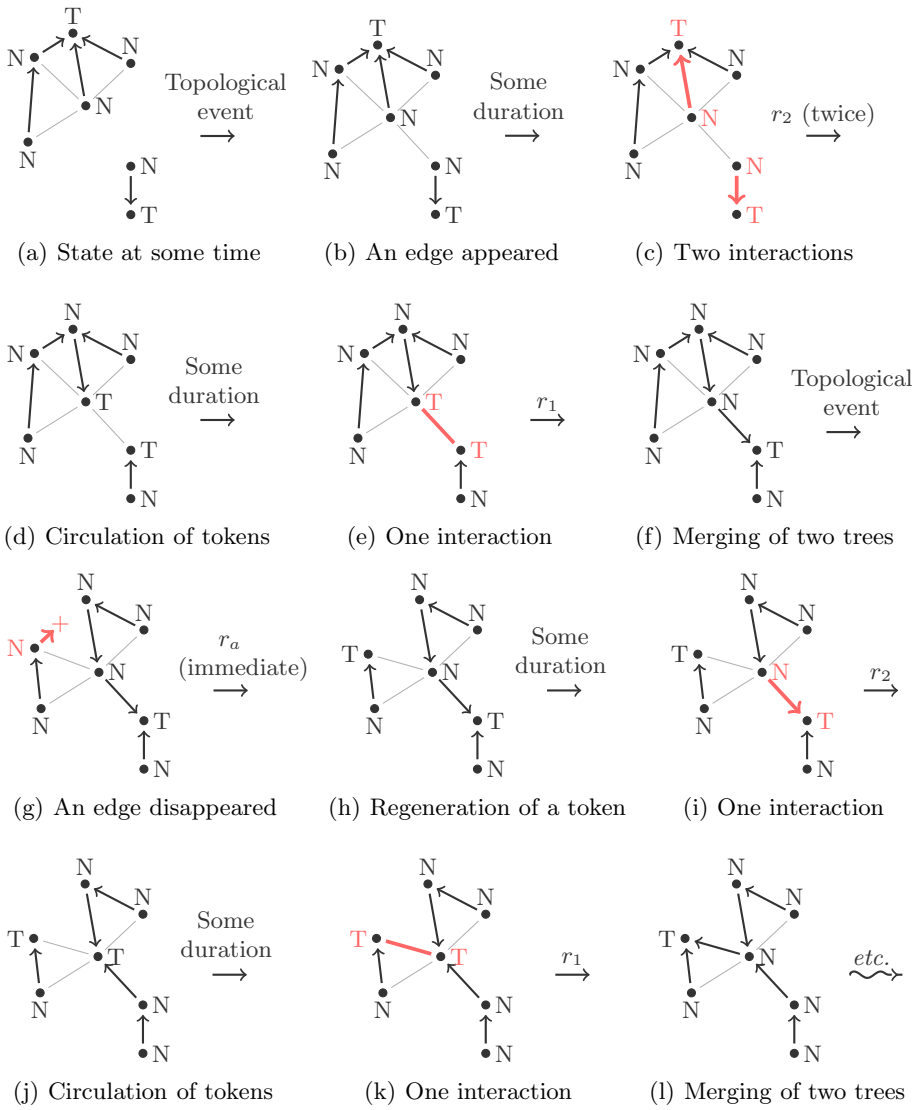
Proof. By Lemmas 1 and 2. □

Theorem 2. *At any time, the trees are cycle-free.*

Proof. The property holds initially. The only way an edge can be added to a tree is by means of applying r_1, which involves two tokens. By Lemma 2, there is at most one token per tree, thus at most one application of r_1 can occur at a time for a given tree, and the two tokens must belong to different trees. □

4.4 Discussion

The algorithmic principle introduced here is very general. In particular, the correctness of the properties we have considered so far does not depend on the order in which the edges are selected for interaction, nor whether some interactions should be favored over others (e.g. r_1 over r_2).



(a) State at some time (b) An edge appeared (c) Two interactions

(d) Circulation of tokens (e) One interaction (f) Merging of two trees

(g) An edge disappeared (h) Regeneration of a token (i) One interaction

(j) Circulation of tokens (k) One interaction (l) Merging of two trees

Fig. 4. A possible sequence of execution of the spanning forest algorithm.

On the other hand, these aspects can have a tremendous impact on the ability of the trees to merge with each other and converge towards a single tree per connected component (remind that the network is expected to be partitioned in general).

Priority among rules: In general, given two neighbor nodes at a given time, there might be more than one eligible rule. This was not the case with this algorithm, since r_1 and r_2 have two incompatible guards (preconditions). However, the matter is worth being discussed. Priority among the rules could be understood in a *weak* sense, enforcing the fact that a rule should not be applied by *these two* nodes if they are able to apply another rule first. Another, much *stronger* sense of priority consists in forbidding a node to apply a given rule as long as another rule is applicable *with any* of its neighbors.

Clearly, in the case of the spanning forest algorithm, merging should be preferred over circulation whenever possible. Enforcing strong priority would thus come to forbid the application of r_2 whenever r_1 can be applied. This behavior is expected to produce larger trees, but at the cost of a strong constraint on the scheduler (probing the state of an entire neighborhood prior to interaction). Without speculating on finer-grain implementations of our principle – which is not the object of this paper – we believe a strong priority mechanism remains somewhat natural in a wireless environment, where nodes routinely broadcast their state to all neighbors, in particular if we assume a synchronous communication model such as $\mathcal{LOCAL}$ or $\mathcal{CONGEST}$ [12].

Role played by both nodes in an interaction: The reader may have noticed that, in the definition of the circulation rule r_2, the guard of the rule is not tested on both sides. That is, u implicitely plays the role of the left node, and v that of the right node. As far as the present work is concerned, we do not want to impose a preferred way to solve this question, as it does not affect correctness. As a suggestion, the scheduler may select edges in a directed way (with a left node, and a right node), or the second direction systematically when an edge is selected and the rule is not applicable in the first direction.

High-level view of the process: Assuming the token has equal probability to move to each neighbor (in the tree), we can regard the circulation as a random walk in the tree. Further, if we assume *strong* priority enforcement between r_1 and r_2, the circulation and merging processes turn into a specific variant of coalescing random walks [7]. This point of view is the one we consider in the next section.

5 Preliminary analysis

In this section, we study the question of how frequent the mergings are. We only provide preliminary results and some thoughts about the complete analysis of this process (which is far beyond the scope of this paper). Hence, we characterize the number of token moves expected in a stationary regime, before a merging

occurs between two given trees in a static context. This value is given as a function of their size and the number of edges connecting them (called *bridges*).

5.1 Random walks in trees

For the sake of analysis (and with loss of generality), we look at the process of merging and circulating tokens as a system of particles that perform random walks *in trees* and coalesce whenever they *meet*. Here, the concept of meeting between two particles is defined with a special meaning. Indeed, in most coalescing particle systems, two particles are said to meet if they happen to be located at a same node, whereas in our case, they meet if they are located at both endpoints of a same edge (remind that the tokens cannot travel beyond their trees).

5.2 Bridges

Given two different trees $\mathcal{T}_1$ and $\mathcal{T}_2$, there may be some edges whose endpoints lie in $\mathcal{T}_1$ on one side, and $\mathcal{T}_2$ on the other side – we call such edges *bridges*. Figure 5 shows an example of two trees that share four bridges.

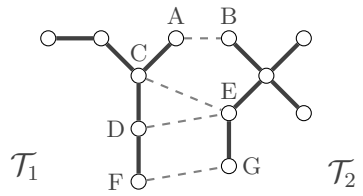


Fig. 5. Example of two trees sharing four bridges (dashed lines).

As discussed in Paragraph 4.4, the enforcement of a *strong* notion of priority between merging and circulation allows one to assume that if two tokens are located on a same bridge, then merging occurs. (This is at least true in the case of two trees, which is the one addressed here.) Hence, the probability that merging occurs is that of having both tokens located at a same bridge.

Let us denote by $Bridges(\mathcal{T}_1, \mathcal{T}_2)$ the set of edges (u, v) such that $u \in E_{\mathcal{T}_1}$ and $v \in E_{\mathcal{T}_2}$. The probability that $\mathcal{T}_1$ and $\mathcal{T}_2$ merge at a given time is equal to:

$$P_{merge(\mathcal{T}_1,\mathcal{T}_2)} = \sum_{(u,v)\in Bridges(\mathcal{T}_1,\mathcal{T}_2)} P[\lambda(u)=T \wedge \lambda(v)=T]. \quad (1)$$

5.3 Probability of being located at a node

In a stationary regime, the probability for a token to be located at a given node v in a graph G (tree or not) is a well-known result in random walk theory, which only depends on the ratio between the degree of v, $d_G(v)$, and the sum of all degrees in G.

In a tree $\mathcal{T}$, the probability a node v hosts the token is thus

$$P(\lambda(v) = T) = \frac{d_{\mathcal{T}}(v)}{2|E_{\mathcal{T}}|} \tag{2}$$

where $|E_{\mathcal{T}}|$ is the size of $\mathcal{T}$. Keep in mind this value corresponds to the stationary regime (when the probabilities no more depend on the initial configuration).

5.4 Expected merging time in the stationary regime

We are interested in the mean number of steps (token moves) required to merge the two trees, assuming the walks are in a stationary regime. Moreover, as trees are bipartite graphs, if both tokens move synchronously it may happen, depending on their initial position, that they never meet. Thus, we assume here that the moves are asynchronous (i.e., one at a time). Equations 1 and 2 allow us to state that the probability for two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ to merge at any step is

$$P_{merge}(\mathcal{T}_1, \mathcal{T}_2) = \sum_{\{(u,v) \in Bridges(\mathcal{T}_1, \mathcal{T}_2)\}} \frac{d_{\mathcal{T}_1}(u)}{2|E_{\mathcal{T}_1}|} \times \frac{d_{\mathcal{T}_2}(v)}{2|E_{\mathcal{T}_2}|} \tag{3}$$

which in turn gives the *expected merging time* in number of steps, as $E_{merge}(\mathcal{T}_1, \mathcal{T}_2) = P_{merge}(\mathcal{T}_1, \mathcal{T}_2)^{-1}$.

However limited, a quick look at these results teaches us some preliminary facts. First, the merging time of two trees of size n in which the nodes degrees d are fairly distributed is in $O(\frac{n^2}{nbBridges \cdot d^2})$. The d^2 term could actually be omitted if we consider that degrees are bounded by some constant (a fair assumption in most wireless networks). Whence a time of $O(\frac{n^2}{nbBridges})$ steps. Whether this time is linear or quadratic in the sizes of the trees depends on the number of bridges (e.g. merging time is linear if the number of bridges is in $O(n)$; it is quadratic if that number is constant; etc.). That in turn, depends on the networking scenario which is considered, and in particular, what *mobility model* is used.

A deeper look is required to understand the behavior of this process. We expect it to be quite difficult to analyze in the general case. Not only the algorithm involves much more than two trees in general, but it is intended to run over highly dynamic topologies, where splittings and mergings occur concurrently. In fact, the metric of interest might be different than convergence time, since the merging process is never expected to converge. A better metric here could be the average number of trees per connected component in a stationary regime.

6 Conclusion

This paper proposed a new mechanism for building and maintaining a forest of spanning trees in highly dynamic networks. The originality of the approach is that the construction is a perpetual ongoing process that takes place at the same time as the trees are used. The principle is very general and relies on token circulation techniques that turns *splittings* and *mergings* of the trees into purely

localized phenomenons. After presenting the algorithm using a coarse grain interaction model, we provided some preliminary observations on the analysis of the corresponding process, regarded for the occasion as a system of coalescing random walks. A deeper analysis of this process is still far from reach, and we expect it to be technically challenging in the general case. As the process is never expected to converge, completion time is not the most relevant metric here (however its characterization in a static and connected context might already be very insightful). Characterizing the average number of trees per connected component in the stationary regime seems to be the relevant metric.

Besides analysis, an avenue of research is to transpose the algorithm into finer-grain communication models. We believe this can be done, at least in synchronous message passing models. Finally, de-randomizing the way tokens circulate (e.g. using Propp machine-like mechanisms) may lower the cover time, and possibly, speed-up the merging process. These questions are open.

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. D.J. Aldous. The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM J. Discret. Math.*, 3(4):450–465, 1990.
3. D. Angluin, J. Aspnes, Z. Diamadi, M. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
4. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *Proc. of the 25th ACM symp on Theory of computing (STOC)*, pages 652–661, New York, USA, 1993.
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. J. Burman and S. Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *Proc. of 21st symp. on Distributed Computing (DISC)*, pages 92–107, 2007.
7. C. Cooper, R. Elsässer, H. Ono, and T. Radzik. Coalescing random walks and voting on graphs. In *Proc. of the 31st ACM symp. on Principles of distributed computing (PODC)*, pages 47–56, 2012.
8. K. Fall. A delay-tolerant network architecture for challenged internets. In *Proc. of Int. Conf. on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 27–34, 2003.
9. A. Ferreira. Building a reference combinatorial model for MANETs. *IEEE Network*, 18(5):24–29, 2004.
10. F.C. Gaertner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Technical report, EPFL, 2003.
11. I. Litovsky, Y. Métivier, and E. Sopena, Graph relabelling systems and distributed algorithms, In *Handbook of graph grammars and computing by graph transformation*, volume III, pages 1–56, World Scientific Publishing, 1999.
12. David Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, 2000.