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-00376701v1

HAL Id: hal-00376701
https://hal.science/hal-00376701v1

Submitted on 20 Apr 2009 (v1), last revised 22 Jul 2013 (v2)

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[2], Frédéric Guinand[3] and Yoann Pigné[3]

[1]SITE, University of Ottawa
800 Avenue King Edward
Ottawa, Ontario K1N 6N5
Canada

[2]LaBRI, University of Bordeaux 1
351 cours de la Libration
33405 Talence cedex
France

[3]University of Le Havre
LITIS EA 4108
BP 540, 76058 Le Havre cedex
France

Abstract— **This paper investigates the problem of building and maintaining distributed spanning trees in dynamic networks. Contrarily to previous solutions, we do not assume the existence of stabilization periods between topological changes, and address the more general case where such changes may occur at anytime and disconnect the network. Hence, we present an algorithm that relies on a perpetual alternation of *topology-induced splittings* and *computation-induced mergings* of a forest of spanning trees, using random walks of tokens. The original idea behind this algorithm is simple: each tree in the forest hosts exactly one token, whose circulation is strictly limited to the edges of the tree. When two tokens meet, the trees are merged and one of the two tokens is destroyed. When a link is broken, the adjacent node, belonging to the token-free tree, generates a new token. The main features of this approach are that both *mergings* and *splittings* are purely localized phenomenon, which allows a transparent and continuous use of the involved subtrees (as far as no higher-level communication is concerned). The algorithm presented here, while briefly introduced in another context, was never analyzed nor properly discussed. We do both here, and provide analytical expressions of the expected merging time of two given trees. We finally propose a substantial optimization to the algorithm that consists in using a *memory-based* bias in the token walks. The impact of this optimization is investigated both analytically and experimentally.**

I. INTRODUCTION

Spanning trees are essential components in communication networks. The availability of such structures simplifies a large number of tasks, among which broadcasting, multicasting, electing, or naming. The computation of spanning trees is therefore a classic problem in distributed computing. From a distributed point of view, constructing a spanning tree implies the collaboration of neighboring nodes to establish relations among some of their common links so that the collection of these links forms a tree that connects them all. In static networks, there is generally a distinction between the construction of a tree and its effective use, both taking place at a different time.

The present document is a working paper whose purpose is to summarize the results we obtained so far on this problem. The reader is therefore invited to check if a later, or published, version of this paper has been made available since April 2009. The work presented here was supported by the *Agence National pour la Recherche*, within the SARAH project, on contract ANR-05-SSIA-0002-01.

Considering spanning trees in truly dynamic networks is slightly different. Indeed, because links are frequently made unavailable, the construction of the tree may not be durably achievable and rather be considered as a continuously running process. Moreover, in some kinds of emerging networks, such as *Delay-Tolerant Networks* [Fal03], the constant end-to-end connectivity of the network cannot be assumed, and the possibility of a single spanning tree covering it may not even exist. It appears however that most of the works in this domain considered the approach of adaptating static network algorithms to the dynamic context, still assuming that constructing a single tree can be achieved during some topologically stable periods.

In this paper we focus on dynamic networks where no stable period can be assumed. As a consequence, the construction of the tree(s) must be seen as a continuous, never-ending process, during which topological events frequently occur. The main idea behind the algorithm studied here, firstly introduced in [Cas06], relies on an entangled alternation of *computation-based mergings* and *topology-based splittings* of a forest of trees. The core mechanism is based on the circulation of several tokens whose number is maintained at exactly one per tree. The fact that each circulation is strictly limited to the edges of the corresponding tree, and the fact that no unique identifiers are required, allows *mergings* and *splittings* to be handled in purely decentralized and localized fashion, without requiring any further communication. Also, the algorithm requires no additional memory than what is needed for the strict encoding of the tree (two bits per edge, and one per vertex). Later on, we propose an optimized version where the circulation of the token is constrained by its past movements. This optimized version requires slightly more memory (from 1 to n additional bits per edge, where n is logarithmic with the degree of the corresponding endpoint vertices). No global knowledge is required for both versions. While offering interesting properties in really dynamic networks, this algorithm is however expected to perform quite poorly in a static network with identities (compared to what a dedicated algorithm could do in that context). It does not address the problem of minimizing the sum of edge costs neither, as equal weights are assumed on all edges.

After reviewing some relevant existing work in Section II and giving preliminar information in Section III, we present

and discuss the original version of the algorithm in Section IV. Section V is then devoted to its analysis using random walks properties. In particular, we give an analytical expression of the *expected merging time* of two trees. In Section VI, a substantial improvement of the algorithm is proposed based on memorizing, locally to each vertex, the n last visited edges so that the token exploration is timely balanced. We conclude in Section VII with some avenues for further research.

II. 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. Providing a comprehensive review of the domain is difficult, if not impossible, especially because the problem was studied by different communities (self-stabilization, random walk, distributed computing) using different paradigms and terminologies (*e.g. token, mobile agent, random walk, legal state, stabilization time, merging time, cover time, tree, forest, etc.*). In spite of this complexity, we attempt to review below the most relevant concepts and approaches to solve this problem.

1) 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 of the self-stabilizing algorithms proposed so far for the spanning tree problem, and the one with the smallest stabilization time was introduced in [AKM$^+$93] (as a coarse-grain graph algorithm, more recently translated into the message passing model in [BK07]). We refer the reader to [Gae03] for a more general survey on self-stabilizing spanning tree algorithms. However, self-stabilizing algorithms assume that no additional faults occur during the stabilization period, which may not be realistic in unpredictable dynamic networks. In contrast, the algorithm studied in the present paper has a stabilization time of one operation, while considering a weaker meaning of what is a legal state (*i.e.,* covering the whole network with a single tree is not part of the condition).

2) random walk: a random walk is a sequence of vertices such that each vertex in the sequence (except the starting vertex) is randomly selected among the neighbors of its predecessor. The expected number of steps required to cover the whole network is called *cover time*. Random walks have been used to solve several problems in distributed systems, such as leader election, token management, and spanning trees. The idea of using random walks to compute spanning trees was first proposed in [Ald90], where a single random walk is considered. Anytime, the set of all covered vertices, along with the edges from which they were visited the first time, defines a random tree that spans the nodes already visited.

3) mobile agents: mobile agents are entities that can "physically" travel across the network, and perform tasks directly on the nodes. These agents may or may not carry their own memory, and adopt a variety of strategies to move within the network. In [BFG$^+$03], distributed random walks of mobile agents (called *tokens* in the paper) were 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 vertex that has 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 similar approach was presented more recently in [AMZ06], where mobile colored agents (equivalent to tokens) construct subtrees in a distributed way. These 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 vertices and eventually destroys the other agent. The advantage of this gradual process is that it avoids the wave propagation. However, unique identifiers are required to generate the colors. To regenerate agents after topological changes, the authors assume an upper bound in the cover time. If a node was not visited during this period, the process concludes that the token was lost and regenerates a new one. The problem is that relying on the cover time is not totally safe. Indeed, a token might stay longer than expected in some part of the tree (unless a very large value is considered). This mechanism furthermore implies that the nodes know an upper bound on the order of their tree (or at least of the entire graph) in order to estimate the cover time. Also, if the rate of topological events is higher than the expected cover time, then the trees are never used.

In comparison to these approaches, the one we propose and study here neither requires any stable periods, nor unique identifiers or wave mechanisms. This is, to the best of our knowledge, the first such attempt.

III. Network model and assumptions

Dynamic networks are generally represented by dynamic graphs. However, as far as we investigated the problems presented next, we did not specifically required a dynamic graph model. At a given moment, the network is therefore represented by an undirected simple graph $G = (V_G, E_G)$, where V_G stands for the set of nodes and E_G stands for the set of communication links available between them. Vertices adjacent to a same edge are said *neighbors*. The set of neighbors of a vertex v is noted $N(v)$. At anytime t, the state of the network is given by a *labelling* on the corresponding graph. More precisely, every vertex $v \in V_G$ is associated with one label $\lambda_t(v)$ representing its algorithmic state, and another label $\lambda_t(v, e)$ for each of its adjacent edges. Each edge is thus labelled on both endpoints, with possibly different values.

The algorithm given in this paper is a coarse-grain atomicity algorithm, which basically means that the communication

model (*e.g.* mailbox, shared memory, or message passing) is abstracted by atomic operations occuring possibly simultaneously on several neighbor vertices (and related edges).

Hence, the algorithm will be described using graph relabelling operations, that are pairs of label patterns $(precondition, action)$, also called *relabelling rules*, which define how the states of neighboring vertices are to be modified [LMS99]. For example,

$$\big(\underline{\text{precondition:}}\ \lambda(v) = \texttt{informed} \wedge \exists v' \in N(v) \mid \lambda(v') = \neg\texttt{informed}, \\ \underline{\text{action:}}\ \lambda(v') := \texttt{informed}\big)$$

represents a propagation of information in the network. When this is non ambiguous, an equivalent graphical notation can be used (*e.g.* inf ¬ inf $\longrightarrow$ inf inf).

A complete algorithm is then given by a (possibly ordered) set of such rules. The fact of using *ordered* rules means that a rule r_i can be locally applied only if no rule $r_{j<i}$ are locally applicable. Note that guaranteeing this property requires to consider the whole neighborhood of the involved vertices before each rule application. We concede that this may complicate the translation of such coarse-grain algorithm into a real communication model.

IV. The spanning forest algorithm

This section presents the spanning forest algorithm and discuss some of its main properties.

A. The locality criterion

Let us consider the scenario depicted Figure 1, where two nodes, A and B, are to decide whether their common edge should be used to merge their respective trees. This scenario illustrates some design choices of the algorithm.

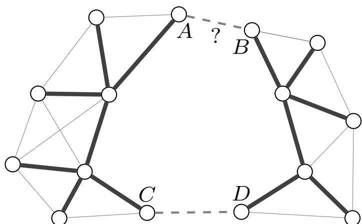

Fig. 1. An example scenario.

In order to take the decision, two problems must be solved:

1) Do A and B effectively belong to different trees, or is there a path linking them within the tree?
2) How to guarantee that no other merging operation simultaneously occur between these two trees (*e.g.* if C and D were performing the same operation in parallel)?

The second problem implies either that i) before merging, a vertex knows that it is the only one capable of doing so in its tree at this time, or that ii) it initiates a consultation in its tree to get the permission from the other vertices or from a central authority (typically the root of its tree). In really dynamic networks, where topological events are expected to occur frequently, raising a consultation within the tree is not conceivable since this may lead to the inefficiency of the process, especially in case of multiple demands. It appears a better option to allow decisions to be made locally by de facto having only one node being able to merge at a time. As a by-product, this also solves the first problem, since two vertices cannot simultaneously have the merging faculty if they do not belong to different trees.

B. The Algorithm

The algorithm, given Figure 2, is based on three operations on tokens: *circulation*, *merging*, and *regeneration*, which aim at maintaining always one, and exactly one, token per tree. Initially, every vertex is a one-vertex tree that has its own token (label T). When two tokens happen to be at distance 1 (hosted by neighboring vertices), they are merged into one single token (the other node is relabelled N), and the corresponding edge is marked as a *tree edge* (rule r_3) by using a different label on each side (1 and 2) to reflect the orientation induced by the remaining token (see below). When no merging is locally achievable, the token is transmitted to any neighbor <u>in</u> the tree (rule r_4), and the orientation mark is updated consequently.

<u>initial states:</u> T for every vertex, ∅ for every edge extremity.

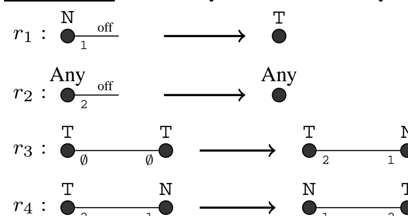

Fig. 2. The spanning forest algorithm, given as a set of relabelling rules.

The key point here is that the labels of the edges always define an oriented tree that is rooted in the vertex that hosts the token. Thanks to these labels, every N-labelled vertex has a unique 'outgoing' edge (label 1), which indicates the direction towards the token. If such an edge is broken for any reason, them this node is thus aware to be now the top most vertex in the hierarchy of its orphan part of the tree (this property remains true whatever the number of simultaneous edge failures). This vertex then simply *regenerates* a new token (rule r_1). On the parent side side, this induces only to 'remove' the local state of the broken edge (rule r_2), unrespectfully to the fact that this node has the token or not. In order to enforce potential mergings, we consider the rules as ordered (that is, r_4 cannot be applied if r_3 is applicable), which implies, as previously mentioned at the end of Section III, that a node consider its whole neighborhood before applying a rule. In some particular configuration however, a few possible mergings may be missed (*e.g.* when a T-labelled vertex has more than one T-labelled neighbors at a time).

V. Analysis

This section studies the question of how frequent the mergings are. In particular, we characterize the expected number of token moves before a merging occurs between two given trees,

as a function of their orders and the number of their shared links. We hope this may later help characterize the expected size of the trees according to properties on the topology dynamics.

A. Asynchronicity

The system is considered asynchronous in the sense that no global clock is available and the distributed operations do not necessarily occur at the same frequencies everywhere in the network. Previous work in the area of local computations considered underlying localized procedures to determine how vertices choose each other to collaborate (*e.g.* [MSZ03]), with the side effect that vertices become somehow synchronized (confined into *rounds*). Since all distributed operations in our algorithm involve at least one T-labelled vertex, we do not use these general-purpose procedures, but rather assume that collaborations are to be initiated by T-labelled vertices only. This makes it possible to consider that each token circulates independently from the others, and consequently that every move of every token constitutes a distinct *step* in the global system.

B. Bridges

Given two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ in the same graph G, there might exist some edges whose extremities belong to $\mathcal{T}_1$ on one side, and $\mathcal{T}_2$ on the other. Let us call such edges *bridges*. After each individual token move, the probability that the two trees merge is equivalent to the probability for their two tokens to be located on the extremities of a same bridge.

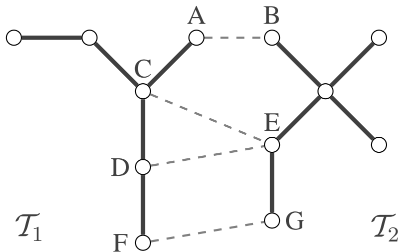

Fig. 3. Example of two trees in the same graph. The bridges between them are represented by dashed lines.

More formally, 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 step s is thus equal to:

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

C. Token circulation vs. random walks

Assuming that a T-labelled vertex has equal chances to apply the rule r_4 with any of its *child* (this can be an implementation choice), the circulation of each token becomes a random walk in its tree. Now, the probability for a random walking token to be positioned on a given vertex v in a graph G (tree or not) is a well-known result that only depends on the degree of v, noted $d_G(v)$. At anytime, for any vertex v in a tree $\mathcal{T}$, the probability that v hosts the token is:

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

In fact, these values hold only after a certain time of circulation, before which the probabilities actually depend on the starting vertex. In more technical terms, this random walk can be seen as a *Markov chain* whose *stationary* (or *equilibrium*) *distribution* corresponds to the values of Eq. 2. Depending on the desired precision, this equilibrium can be considered as reached after a certain time of circulation, which is called *mixing time*. As discussed later on in this paper, this time has an important impact on the algorithm performance.

D. Expected merging time

The *expected merging time* constitutes an estimation of the mean number of steps (number of token moves in our context) required to merge the trees. Let us first assume that the *mixing time* is instantaneous, that is, the probabilities of presence of the token are always equal to those of Eq. 2 for every vertex. This assumption will be released in Section VI, where we explore the impact of this time on the performances of the algorithm. Hence, considering Eq. 1 together with Eq. 2, the probability that two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ merge at a given step is:

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

which in turn gives the *expected merging time* (in number of steps), as:

$$\begin{aligned} E_{merge}(\mathcal{T}_1, \mathcal{T}_2) = &\; (P_{merge}(\mathcal{T}_1, \mathcal{T}_2))^{-1} \\ = &\left(\sum_{\{(u,v)\in Bridges(\mathcal{T}_1,\mathcal{T}_2)\}} \frac{d_{\mathcal{T}_1}(u)}{2|E_{\mathcal{T}_1}|} \times \frac{d_{\mathcal{T}_2}(v)}{2|E_{\mathcal{T}_2}|} \right)^{-1} \end{aligned} \quad (4)$$

E. Further analytical results

We have considered here the merging probabilities and expected merging time of two static trees. While this provides a good intuition of the performance of the algorithm, this is not the end of the story. First of all the algorithm is intended to run on dynamic topologies. The most interesting parameter is thus the average number and order of trees one may expect by running this algorithm in a given mobility scenario (*e.g.* given an expected rate of topological changes). This study will most likely require Eq. 4 as a first step, though. Also, an intermediate step will certainly be to generalize it for more than two trees and potential concurrency between their mergings.

VI. Optimization

This section presents a substantial optimization of the algorithm, based on introducing the use of memory in the random walk in order to reduce its mixing time.

A. *Motivations*

To compare the expected time of Eq. 4 with simulation results, we experimentally measured the merging time of pairs of fixed order random trees (that were randomly generated). Once a pair of trees built, the test consisted in adding three random edges between them (bridges), and measuring the number of steps required to merge thereafter. Two scenarios were tested, with different orders for the trees. In the first scenario, one of the trees was of order 20, the other 8. In the second scenario, both trees were of order 12. According to Eq. 4, the expected merging time for the first scenario should have been:

$$
\begin{aligned}
E_{merge}(\mathcal{T}_1,\mathcal{T}_2) &= \Big(\sum_{1..3}\big(\tfrac{av.deg(\mathcal{T}_1)}{2|E_{\mathcal{T}_1}|} \times \big(\tfrac{av.deg(\mathcal{T}_2)}{2|E_{\mathcal{T}_2}|}\big)\Big)^{-1}\\
&= \Big(\sum_{1..3}\big(\tfrac{|E_{\mathcal{T}_1}|/|V_{\mathcal{T}_1}|}{2|E_{\mathcal{T}_1}|} \times \big(\tfrac{|E_{\mathcal{T}_2}|/|V_{\mathcal{T}_2}|}{2|E_{\mathcal{T}_2}|}\big)\Big)^{-1}\\
&\simeq 53.3 \text{ steps.}
\end{aligned}
$$

and 48 steps exactly for the second scenario. For both scenarios, we ran 3000 iterations of the test, each one executed on different random trees. The average merging time we observed was of 121.0 steps in the first case (instead of 53.3), and 104.2 in the second case (instead of 48). This difference means that the probabilities of token presence among the nodes was not effectively distributed as predicted by Equation 2. In other words, the equilibrium distribution cannot be considered as *reached* as soon as the walk has started, and this has an impact on the performance.

More intuitively, this problem can be thought of as the token going back and forth within a small subset of the vertices, and taking a relatively long time to visit other new nodes. If the number of nodes involved in a bridge is small in comparison to the total number of nodes in the tree, then this obviously increases the average time needed to reach such nodes (and consequently the merging time). In terms of random walks, we believe that the *mixing time* has an impact on the *hitting time*, and that this impact is as negative as the relative number of bridge nodes is small in the trees. Note that as far as this paper is concerned, these statements are still to be formally investigated.

B. *A Memory-based approach*

Based of the above observation, we propose an optimization that intends to reduce the *mixing time* in order to increase the probability to visit minorities of vertices more quickly. The improvement consists in memorizing, locally to each vertex v, the n most recently visited incident edges so that the token is not sent back on them, but preferably on any of the $d(v) - n$ other edges (or on the least recently visited one if $d(v) \leq n$). The chosen value for n actually controls the level of randomness of the walk, 0 being equivalent to the original version, and $max(d(v) : v \in V_{\mathcal{T}}) - 1$ being equivalent to a deterministic token circulation.

In order to measure the impact of the memory level n on the mixing time, we considered single random trees of order 20 and observed how tokens circulate in them depending on the value of n. More precisely, we measured the difference between the effective distribution of token presence on vertices and the values given by Equation 2 The results averaged over 1000 simulations (with different initial graphs for each run) are presented Figure 4.

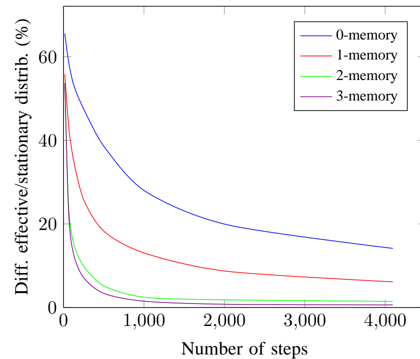

Fig. 4. Impact of the level of memory on the mixing time towards the values of Equation 2.

As expected, the mixing time decreases as the level of memory increases. It can also be noted that all versions well and truly converge towards the values of Equations 2 (which fact was not necessarily obvious at first sight).

Now, using the same scenarios as considered in paragraph VI-A, we tested the different levels of memory upon the merging time. Figure 5 shows the results, averaged over 3000 iterations for each memory level. As one can see on this plot, these results are clear-cut, and even beyond expectations, since the effective merging time becomes indentical to the expected merging time of Eq. 4 as the level of memory increases.

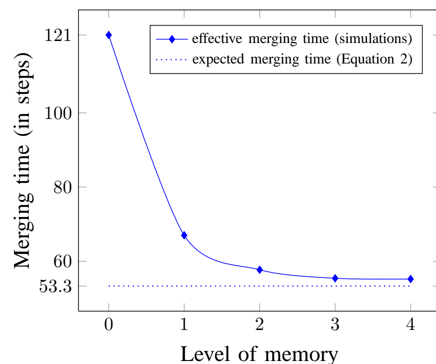

Fig. 5. Impact of the level of memory on the merging time (Scenario 1).

The simulation results for the second scenario, shown in Figure 6, are slightly less obvious to interpret. Indeed, it appears that, in this case, the best level of memory is 1 and that the performance deteriorates for larger values. The reason for this behavior is most probably due to the determinism induced by the memory. Indeed, as the level of memory increases, a cyclic effect on the token circulation will appear, and in case of identical (or *coprime*) orders for the trees, this may put the tokens somehow *in-phase*. If the system were globally synchronous, such effect could in fact generate infinite merging times, which does not happen here. This problem of phase is also likely to be mitigated by the network

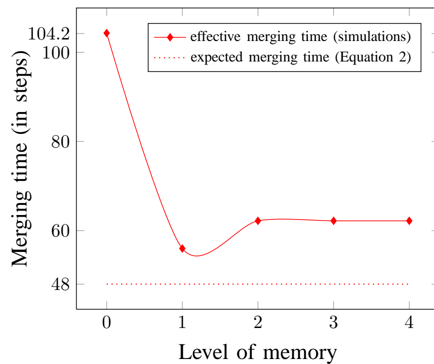

Fig. 6. Impact of the level of memory on the merging time (Scenario 2).

dynamics, whose side effect is to regularly breaks any potential symmetry.

VII. Conclusion

This paper presented several versions of a spanning forest algorithm for dynamic networks. The originality of this algorithm is that the construction (and maintainance) of the trees is a continuous process that takes place at the same time as the tree is used. This property results from a particular random walk technique that turns *splittings* and *mergings* of the trees into purely localized phenomenons. Besides the presentation of the algorithm, this work focused on understanding how some properties of the walk could impact the performance. To start, we characterized the expected merging time of two given trees if the walks were to offer some ideal properties (namely, a *mixing time* of 0). Based on experimental results, we observed that the mixing time had indeed an impact on the ability to reach (or *hit*) the nodes located on potential *bridges* (the higher mixing time, the higher hitting time when these nodes are a minority). An optimized version was then proposed to try to improve the hitting time through the mixing time. This optimization consisted in avoiding the last k locally visited edges during the walk. This technique, that one can see in the spirit of tabu searches approaches, prove very efficient in the tested scenarios (more than halving the average number of token moves required to merge two given trees).

Beyond these analytical and experimental results, a number of questions remain open at this stage of investigation. First, the nature of the relation between the mixing and the hitting time is not totally clear, and an extra work will be required to get deeper insights on this question. A better understanding on these mechanisms could also be obtained by measuring the influence of additional parameters, such as the degree of the bridge nodes or the existence of particular properties on the network geometry (*e.g. unit disc graph*). We hope to be eventually able to characterize the average size of the trees as a function of the density and the rate of topological changes. Another important question is how close the algorithm results are from the optimal solution (that is, the average ratio between the number of trees and the number of connected components). Finally, one may wonder whether this kind of algorithms is suitable to real networking contexts, such as state-of-the-art wireless ad hoc networks, in which the detection of the neighborhood is not instantaneous (nor totally guaranteed), and each token move implicitely generates several exchanges at the lower levels. We are currently performing some real-life experiments to try to answer this question.

References

[AKM+93] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 652–661, New York, NY, USA, 1993. ACM.

[Ald90] D.J. Aldous. The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM J. Discret. Math.*, 3(4):450–465, 1990.

[AMZ06] S. Abbas, M. Mosbah, and A. Zemmari. Distributed computation of a spanning tree in a dynamic graph by mobile agents. *Engineering of Intelligent Systems, 2006 IEEE International Conference on*, pages 1–6, 0-0 2006.

[BFG+03] 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.

[BK07] J. Burman and S. Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *DISC*, pages 92–107, 2007.

[Cas06] A. Casteigts. Model driven capabilities of the DA-GRS model. In *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, pages 24–32, Washington, DC, USA, 2006. IEEE Computer Society.

[Fal03] K. Fall. A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, New York, NY, USA, 2003. ACM.

[Gae03] F.C. Gaertner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Technical report, 2003.

[LMS99] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In World Scientific Publishing, editor, *Handbook of graph grammars and computing by graph transformation*, volume III, Eds. H. Ehrig, H.J. Kreowski, U. Montanari and G. Rozenberg, pages 1–56, 1999.

[MSZ03] Y. Métivier, N. Saheb, and A. Zemmari. Analysis of a randomized rendezvous algorithm. *Inf. Comput.*, 184(1):109–128, 2003.