A Distributed Algorithm for Computing a Spanning Tree in Anonymous T-prime Graphs

Y. Métivier, M. Mosbah, P.-A. Wacrenier

LaBRI,
ENSEIRB / University of Bordeaux 1
France
{metivier|mosbah|wacren}@labri.fr

S. Gruner

ECS / DSSE
University of Southampton
England
sg@ecs.soton.ac.uk

Abstract

We describe a distributed algorithm for computing spanning trees in anonymous networks. The processors communicate with their neighbours by asynchronous message passing. The algorithm is correct and terminates for networks belonging to the class of T-prime graphs. We also present some experimental results concerning the complexity of our algorithm.
Keywords : anonymous network, asynchronous message passing, asynchronous system, election, distributed computations, T-prime graph, spanning tree, spanning forest.

1 Introduction

Computing a *spanning tree* of a network is an important problem in distributed computing. Trees are essential structures in various communication protocols (synchronization, deadlock resolution, information broadcasting). This problem is also closely related to the election problem. The aim of an election in a network of processors is to choose exactly one element in the set of processors : this element is called elected or leader. The leader can be used subsequently to make decisions or to centralize some information. In fact, the problem of finding a leader is reducible to the problem of finding a spanning tree.

In this paper we describe a new algorithm for the distributed computation of spanning trees under the relevant assumption that the network is *anonymous*. Thereby, we consider networks of (nearly) arbitrary topology. Such networks are modelled as undirected connected *graphs*, wherein the nodes represent processors and the edges represent full-duplex communication channels.

Similar solutions are already known for similar problems, see for example [16, 18]. Spanning tree problems have indeed been solved under many different assumptions: the graph can be a ring, a complete graph, or a general connected graph. The underlying communication system may be synchronous or asynchronous. The model graph may be directed or undirected. The processors might have unique identifiers or be indistinguishable. The processors may or may not know the size of the graph, and so on.

The model of distributed computation we deal with is the following. (1) The system has *no global clock*. (2) Communication is based on bidirectional and perfect[1] FIFO-channels. (3) Communication is performed by *asynchronous message passing*. (4) The network's *size is known to each* processor of the network. (5) The network is *anonymous*.
Under these assumptions, we prove that our algorithm is correct if the topology of the network is a T-prime graph (it does not exist a non trivial spanning forest where all trees have the same size).

The intuitive idea of the algorithm is the following. Subtrees are constructed in a distributed way on the graph. Then, each root of a subtree tries to capture an adjacent subtree having less number of nodes. Once captured, both trees merge and the new root is that of the tree with highest size. Of course, nodes of each subtree maintain required information to make the subtree merged with others. Initially, each node of the graph is a subtree. To beak the symmetry, we need two simple probabilistic procedures which are in fact Las Vegas procedures [12]. The first one is used in order to enable a rendez-vous between two adjacent vertices, the second one is used to make an election between two adjacent vertices. Indeed, it is a probabilistic implementation of an initial rule which starts up

[1] Messages are neither lost nor disturbed

the algorithm. Of course, if processors communicate by synchronous message passing, we don't need these procedures [18]. We have implemented this algorithm and performed many experiments. Although the algorithm has been proved to be correct for T-Prime graphs, we made experiments even on non T-prime graphs.

The paper is organized as follows. Section 2 presents related work. Notations and definitions are given in Section 3. In Section 4, we present the algoritm which computes a spanning tree. Experimental results for T-prime and non T-prime graphs are given in Section 5.

2 Related Work

The literature on particular approaches to distributed spanning tree computation under particular assumptions is remarkably rich. Many of those contributions are concerned with *minimum weight* spanning trees, see for example [9]. In the following, we focus our attention on some of those related contributions. In general, the good results (w.r.t. performance) of those approaches are based on global structural assumptions that we did not follow in our approach.

Constraints as *balanced growth* or *dialogue between subtrees* as in [8, 10] do not occur in our approach. Our algorithm is more parallel in the sense that a root L_1 may catch a root L_2, and at the same time L_2 may catch a root L_3 and so on, which means that a root may capture several trees and may be captured itself simultaneously.

In [7] a *self stabilizing* probabilistic method is described to decide if there are several trees: A root discovers that there are more roots with the same maximal ID by repeatedly re-colouring its tree. Each recolouring starts from the root which chooses the new color at random. A vertex detects the existence of another tree by observing that one of its neighbors is not colored neither by the old color, nor by its new. If the same ID appears several times, a random extension of the ID is achieved. This probabilistic method may be applied to our algorithm such that we do not need to know the size of the trees any more. Other approaches based on self stabilization are, for example [15] and [1].

In [6, 11, 17] some *election algorithms* are given for the family of prime rings which are ring graphs having a prime number of vertices. Those correspond exactly to the family of $T-$prime graphs in the case of rings. The algorithms presented in [6, 17] are self-stabilizing.

3 Basic Notions and Notation on Graphs

3.1 Generalities

A *simple graph* $G = (V, E)$ is defined as a finite set V of *vertices* together with a set E of *edges* which is a set of pairs of different vertices: $E \subseteq \{\{v, v'\} \mid v, v' \in V,\ v \neq v'\}$. The cardinality of V is called the *size* of the graph. A *tree* is a connected graph containing no cycles. A subgraph of G is a graph obtainable by the removal of a number of edges and/or vertices of G. The removal of a vertex necessarily implies the removal of every edge incident with it. A connected component of a graph is a maximal connected subgraph. A spanning subgraph of a connected graph G is a subgraph of G obtained by removing edges. A spanning tree is a spanning subgraph which is a tree. A *forest* is a graph whose connected components are trees. A spanning forest of a graph G is a forest which contains the set of vertices of G.

3.2 T-Prime Graphs

Let $G = (V, E)$ be a connected graph of size n, let r be an integer divisor of n (i.e. $n = kr$ for some positive integer k). We say that G is r-factorizable if G admits a spanning forest whose trees are all of size r. The graph G is said to be *T-prime* if it is not r-factorizable for any integer r such that $n > r > 1$. A graph having a prime n size is T-prime. A complete graph of size n or a ring of size n is T-prime iff n is prime. There exist T-prime graphs without prime size, as depicted, for example, in **Fig.1**.

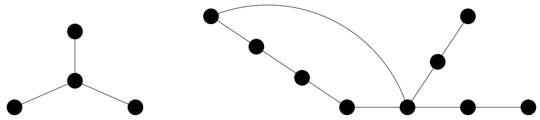

Figure 1: *two T-prime graphs without prime size*

4 The Spanning Tree Algorithm

General Properties. Let G be a graph, let $\mathcal{F}$ be a spanning forest of G, let T_1 and T_2 be two trees belonging to $\mathcal{F}$. T_1 and T_2 are said to be *adjacent* if there exists an edge $\{v_1, v_2\}$ of G such that v_1 is a vertex of T_1 and v_2 is a vertex of T_2. Let n be the size of G. The network be anonymous. Our algorithm has then the following properties:

(A) Each node v of the graph has a *weight* denoted $w(v)$. Initially the weight of each node is 1.

(B) A spanning forest of the graph is maintained. Initially each node of the graph is a (degenerated) tree of the forest.

(C) Each tree of the forest has a node which is a *leader* [18] with a weight equal to the size of its tree, whilst the weight of the other nodes is 0.

(D) Initially, the net contains at least one *initiator*, and each initiator might become the leader of a 2-sized tree connecting this leader together with one single neighbour vertex which has still been "alone" so far. In such case, the leader's weight increases to 2 whilst the weight of the neighbour vertex decreases to 0.

(E) A leader L of a tree, such that $w(L) < n$, tries to find an adjacent tree having a leader L' such that $w(L) > w(L')$.

(F) When a leader L has found an adjacent tree with leader L' such that $w(L) > w(L')$, then the two trees are merged into a new tree such that L remains the leader with a weight in the size of the new tree, thus $w(L)_{\text{new}} := w(L)_{\text{old}} + w(L')_{old}$. The weight of the "defeated" former leader L' decreases to $w(L')_{\text{new}} := 0$.

As a consequence it turns out that if G is a $T-$prime graph, the distributed algorithm described above computes a spanning tree in G. Moreover, our algorithm is also a distributed election algorithm: the elected vertex is the root of the spanning tree.

Description of the algorithm. Two main states have to be distinguished for each process: **root** and **node**. A node p must have exactly one father and may have sons: the array $status_p$ holds this relation. Via the variable $weight_p$, a root p (which is a process without father) knows a lower bound of the size of its own tree. The goal of a root is to conquer neighbouring trees. For this purpose, a root sends *exploration messages* to its neighbours. These exploration messages contain the current weight of the tree belonging to the sending root.

Exploration messages travel down from the root, and each node of the tree forwards the messages to its next neighbours. When a node receives an exploration message from a neutral neighbour (which is neither its father nor one of its sons) it transmits the message to its father. A root is captured when it receives an exploration message with a strictly greater weight than its own. In this case, the captured root sends its own weight to the stronger root which has transmitted the exploration message. Then, the captured weaker root becomes an ordinary node of the dominant tree. The weight of the captured sub-tree is added to the bigger tree via the path of the greatest foreign exploration message which is memorized by the visited tree.

The following variables encode a directed graph:

- $state_p$ is either equal to root or node
- $status_p[q]$ is an array which, for each neighbour q of p, contains the fact that q is either the **father** of p, either a **son** of p or a **neutral** neighbour. (The variable $daddy_p$ denotes the local name of the father of p.)

The following variables are used to establish a rendez-vous and do the "kick-off":

- $waiting_p$ is false if and only if p has established a rendez-vous
- r_p (the local name of a neighbour) designates the neighbour, randomly chosen by p in order to establish the rendez-vous
- b_p is a bit randomly chosen by p for the duel with r_p.

The following variables are used to build a spanning tree:

- $weight_p$ contains a lower bound of the number of nodes of p's tree when p is a root
- $max\text{-}weight_p$ is the greatest weight of the weight exploration messages received by p
- $visited\text{-}from_p$ contains the local name of the sender of the greatest ExploUp message received by p
- $the\text{-}end_p$ is true if and only if $weight_p = graphsize_p$.[2]

Here is the initial variable declaration of our algorithm in pseudo-code:

```
boolean : waiting_p        = TRUE;
integer : b_p, r_p;
status  : state_p          = root;
integer : weight_p         = 1;
integer : graphsize_p;
integer : w;
integer : income;
[status]: state_p[q]       = [neutral,...,neutral];
boolean : the-end_p        = FALSE;
name    : daddy_p;
name    : visited-from_p;
integer : max-weight_p     = 1;
```

[2]Please remember the assumptions that we made above.

Because the initial situation being perfectly homogeneous, we need two *probabilistic* procedures (having LasVegas property) for breaking this initial symmetry.[3] These LasVegas procedures allow for rendez-vous between two adjacent vertices and for performing local elections (duels) between two vertices involved in rendez-vous.

Initially, each process (of weight 1) tries to get in rendez-vous with a neighbour process. In case of success, it proceeds with the following steps:

1. Choose a neighbour k at random
2. Send the choice to k with message $\langle \mathsf{You} \rangle$
3. Await the choice of k and meanwhile refuse any rendez-vous from the other neighbours by sending the message $\langle \mathsf{NotYou} \rangle$

These three steps are iterated by any process p until the chosen neighbour k chooses p or p is captured. Here is the pseudo-code of that procedure:

```
PROCEDURE call
BEGIN
r_p = rand(1,...,|Neigh_p|);
send <You> to r_p;
END
```

After two processes have mutually acknowledged a rendez-vous, they run into a duel “on numbers” whereby each process randomly chooses a bit and sends its value to its opponent. At the same time, it awaits the bit coming from the opponent with message $\langle \mathsf{Choice}, bit \rangle$. This operation is repeated until the two bits have different values. As soon as this is the case, the process p which has chosen the 0 bit looses its root state and becomes a node: its variable $weight_p$ is set to 0, and it memorizes the local name of its father in the array $status_p$. Finally, it sends the $\langle \mathsf{WinningUP}, 1 \rangle$ message to the winner of the duel:

```
PROCEDURE duel(integer : q)
BEGIN
b_p = rand(0, 1);
send <Choice, b_p> to q;
END
```

Please note that a duel may be interrupted by an explorative message sent by another root of weight ≥ 2. For this reason, the winner of a duel must wait for the WinningUP message before incrementing its own weight:

```
EVENT receipt of <You> from q
BEGIN
if (state_p   == root and
    waiting_p == TRUE and
    q == r_p)
 then {waiting_p = FALSE;
       duel(q);}
 else {send <NotYou> to q;}
END

EVENT receipt of <NotYou> from q
BEGIN
if (state_p == root)
 then {call;}
 else {;}
END

EVENT receipt of <Choice, x> from q
BEGIN
if (state_p  == root and
    weight_p == 1)
 then {if (x == b_p)
        then {duel(q);}
        else {if (x == 1)
               then {weight_p    = 0;
                     state_p     = node;
                     status_p[q] = father;
                     daddy_p     = q;
                     send <WinningUp, 1> to q;
                     emitted-to_p[q] = 1;}
               else {;}}
 else {;}
END
```

[3] If the processors would communicate by synchronous message passing, we would not need these initial procedures [18].

After this initial phase —which takes place autonomously and simultaneously everywhere in the network— we now turn to the construction of the spanning tree.

When a root receives a WinningUp message together with the weight of the conquered tree, it updates its own weight and continues with sending further explorative messages in order to conquer other trees. There are two kinds of explorative messages:

- $\langle \mathsf{ExploDown}, w \rangle$: explorative message sent by a root of weight w to every node of its tree. Indeed, when a node receives this message, it transmits it to its sons and sends an $\langle \mathsf{ExploUp}, w \rangle$ to its neutral neighbours.
- $\langle \mathsf{ExploUp}, w \rangle$: explorative message initiated by a process in order to capture neighbouring trees. When a node p receives this message from a process q, it compares w with $max\text{-}weight_p$: either $w \leq max\text{-}weight_p$ then the message is destroyed, or the message is transmitted to the father of p whereby $max\text{-}weight_p$ is set to w and $visited\text{-}from_p$ is set to q for encoding a path from p to the tree which has initiated the ExploUp message:

```
PROCEDURE sendTo(type : recipient, message : msg)
BEGIN
[integer] k_p = [1, 2, ..., Neigh_p];
forall (k_p[i] in k_p)
 do {if (status_p[i] == recipient)
      then {send msg to k_p[i];}
      else {;}}
END

EVENT receipt of <ExploDown, w> from q
BEGIN
if (w >= max-weight_p)
 then {max-weight_p = w;
       SendTo(son, <ExploDown, w>);
       SendTo(neutral, <ExploUp, w>);}
 else {;}
END

EVENT receipt of <ExploUp, w> from q
BEGIN
if (w > max-weight_p)
 then {visited-from_p = q;
       max-weight_p   = w;
       if (state_p == root)
        then {state_p = node;
              if (status_p[q] == son)
               then {send <WinningDown, weight_p> to q;}
               else {send <WinningUp, weight_p> to q;};
              status_p[q] = father;
              daddy_p = q;
              weight_p = 0;}
        else {if (q /= daddy_p)
              then {send <ExploUp, w> to daddy_p;}
              else {;}}}
 else {;}
END
```

Whenever an $\langle \textsf{ExploUp}, w\rangle$ message reaches a root r, it captures this root if and only if $w > weight_r$. Thanks to this restriction, cycles are avoided: two trees of the same weight cannot capture each other simultaneously, and a root cannot capture itself. A captured root r becomes a node, and the message $\langle \textsf{WinningDown}, weigth_r\rangle$ is sent to the greatest detected neighbour tree n detected) via the path encoded by the *visited-from* variables. More precisely:

- $\langle \textsf{WinningDown}, income\rangle$: this message is initiated by a captured root p of weight *income* and sent to the son q which had sent the explorative up message. q becomes the father of the captured root. When a node q receives this message from its father p, it first switches the status of p from father into son. Then, q send a message to the node pointed by $visited\text{-}from_q$, which is
  - either a son of the node: in this case the message will be $\langle \textsf{WinningDown}, income\rangle$
  - or a neutral neighbour: in that case the message will be $\langle \textsf{WinningUp}, income\rangle$

  Then the node pointed by $visited\text{-}from_q$ becomes the new father of q
- $\langle \textsf{WinningUp}, income\rangle$: as we have just seen, this message is initiated by a process q at the frontier of the two trees and sent to its new father p. When p receives this message, its $status_p[q]$ is set to "son". Now,
  - either the process p is a node: in this case it transmits the same message to its father
  - or it is a root: in that case it increments its own weight by *income* and sends further explorative messages in order to conquer other trees.

The algorithm terminates when the weight of a root has reached the known size of the network:

```
EVENT receipt of <WinningDown, rootw> from q
BEGIN
if (status_p[visited-from_p] == neutral)
 then {send <WinningUp, rootw> to visited-from_p;}
 else {send <WinningDown, rootw> to visited-from_p;};
status_p[q] = son;
status_p[visited-from_p] = father;
daddy_p = visited-from_p;
END

EVENT receipt of <WinningUp, income> from q
BEGIN
if (state_p == root)
 then {weight_p = weight_p + income;
       max-weight_p = weight_p;
       status_p[q]  = son;
       if (weight_p == graphsize_p)
        then {the-end_p = TRUE;}
        else {SendTo(son, <ExploDown, weight_p>);
              SendTo(neutral, <ExploUp, weight_p>);}}
 else {if (status_p[q] == neutral)
        then {status_p[q] = son;
              send <WinningUp, income> to daddy_p;}
        else {;}}
END
```

Please note that the processes need to know the size of the network only to detect the termination of the algorithm. In [14] it is shown that there is *no* distributed algorithm which enables an election in the class of T-prime graphs if the size of the graph is unknown. Due to lack of space in this paper, the reader must be referred to the technical report [13] for the correctness proof of our distributed spanning tree algorithm. Finally we can remark that our algorithm may be extended to the case where each node p of the network is endowed with a weight a_p which is in fact a positive integer and the sum of the weights is prime. It may be also extended to the case where the number of edges is prime.

5 Experimental Results

In this section we present some experimental results concerning our distributed algorithm which has been introduced above. The *purpose* of our experiments was to provide us with some first intuition w.r.t. the runtime complexity that our algorithm *could* produce from a (future) theoretical point of view. The interpretation of the results is based on the assumption that the *number of exchanged messages* during the computation is an acceptable measure of the algorithm's runtime complexity.

The execution of these experiments is based on our *tool* [2, 3] for simulating and visualizing the runtime behaviour of distributed algorithms.

5.1 ViSiDiA: A tool for Visualizing and Simulating Distributed Algorithms

ViSiDiA [2, 3, 5] is a tool that has been developed in order to help in the design, the experimentation, the validation and the visualization of distributed algorithms. It is written in Java and it allows the user to model a network, to implement and to execute a distributed algorithm. We deal with distributed algorithms encoded by means of local computations[4].

- Network construction:
  A friendly Graphical User Interface to draw by "drag and drop" a graph which will model the network. The user can add, delete, or select vertices, edges or subgraphs. Visual attributes of vertices and edges such as labels, colors or shapes have default values, but they can be easily customized, for instance to assign an initial labelling to the vertices of the graph, such as a label A to a particular vertex, and N to all other vertices. Note that N is the default label of vertices.

- Algorithm implementation:
  The tool provides a library of high level primitives to program the corresponding local computations[4, 3, 5]. Note that one has to choose first the type of local computation by choosing one of the three primitives:
  - *rendezVous()*: a function that returns the neighbour with whom the synchronization occurs.
  - *starSyncho1()*: returns the center of the star during a star synchronization. Only the center can update its attributes.
  - *starSynchro2()*: returns the center of the star during a star synchronization. The center and its neighbours can update their attributes.

  For the algorithm, we have used the rendezVous synchronization. Moreover, communications between processors can be expressed by primitives such as *sendTo(neighbour, message)*, and *receiveFrom(neighbour)*.

- Execution:
  After compiling the module implementing the algorithm, the user can execute it by pressing on appropriate buttoms provided by the interface. The system automatically creates and assigns to each vertex a java thread which will run a copy (a clone) of the code implementing the algorithm. The user can observe the messages exchanged between vertices (threads), and their states. In particular, label changes of vertices can be seen online. The whole algorithm is animated in such a way that the user can follow its execution. Moreover, the number of exchanged messages is computed and displayed. The tool has also a module to perfom a sequence of expermients[4].

Many distributed algorithms are already implemented and can be directly animated and experimented. These include leader election, spanning tree computation, enumeration and detection of stable properties.

5.2 Experiments

In the context of this paper, experiments concerning two families of graphs are interesting, namely T-prime graphs and, for comparison, also *non-T*-prime graphs: In T-prime graphs, for which success of the algorithm is guaranteed, we are mainly interested in the empirical run-time *complexity* of this algorithm. In non-T-prime graphs, however, for which success of the algorithm cannot be guaranteed in general, our main interest was focussed on the empirical *success quota* of the algorithm.

T-prime Graphs. Inspecting the syntactic structure of our algorithm, we see that the number of exchanged messages will probably depend strongly on the *connectivity* of the input graph. The connectivity $c_G \in \mathbb{Q}$ of a graph G can be defined in the rational numbers as

$$c_G := \frac{|E_G|}{|V_G|}$$

whereby E_G is the set of edges and V_G the set of vertices (nodes) of G. Thus, we may expect the number of exchanged messages to be high in graphs with high connectivity and low in graphs with low connectivity. For this reason, we have performed experiments with a class of graphs with very high connectivity, the *cliques*, and with a class of very low connectivity, the *rings*, in order to find out some upper and lower bounds of the empirical complexity of the SpanningTree algorithm. Thereby it is assumed that "ordinary" graphs with connectivities between those of cliques and rings produce an average complexity which is higher than the one for rings and lower than the one for cliques.

For the family of all *ring* graphs R, the connectivity is small and constant, namely $c_R = 1$. We took prime-rings of sizes 3, 5, 7, 11, 13, 17, 19, 23, and 29. The experiment is repeated five times per ring (to show the LasVegas property of our algorithm). The results of this experiment are reported in **Fig.2** (see Appendix of this paper). We observe a rather modest increase of messages with the increasing size of the input graphs. The empirical average complexity on ring graphs is difficult to estimate with this small amount of data, but it seems to be $\mathcal{O}(|E|^2)$ where $|E|$ is the size (in number of nodes) of the input graph.

For comparison we now regard an analogous experiment with prime-*cliques* instead of rings. We take cliques of sizes 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29 (in number of nodes). The according numbers of edges is determined by

$$|E| = \frac{k(k+1)}{2}$$

where $k = |V| - 1$ according to the well-known chain-sum formula for $i = 1 + \cdots + k$. Thus, for the family of all cliques C, the connectivity is

$$c_C = \frac{k(k+1)}{2} \cdot \frac{1}{k+1} = \frac{k}{2} \; .$$

Again, we count the number of exchanged messages m until success, and we repeat the experiment 5 times per graph. The results of this experiment are shown in **Fig.3** (see Appendix). In contrast to the previous experiment, we observe a rapidly increasing number of exchanged messages. As sketched in **Fig.4** (see Appendix), we might estimate an empirical average complexity of $\mathcal{O}(m) \approx 10 \cdot \mathcal{O}(|E|)$ where E is the set of edges of the graph.

As shown in Section 4, in-tree communication flows only along the branches of the spanning tree under ongoing construction. Perhaps the performance of this algorithm could still be somewhat improved if also the other edges of the graph were used for in-tree communication.

Of course, neither ring graphs nor complete graphs (cliques) are typical examples of computer network topology. However, we have studied these cases as *limit cases* with respect to the property of connectivity. Assuming that the connectivity of no real-world computer network is lower than ring connectivity or higher than clique connectivity, we might reckon that the runtime complexity of our protocol, applied to real-world networks, will not be better than ring complexity and not worse than clique complexity. Further studies will be carried out to refine these coarse assumptions.

Non-T-prime Graphs. We now focus on the success rate of this algorithm under the precondition that failure is possible: we conjecture that the success quota of our algorithm converges to 100 per cent with the size of the non-T-prime input graphs increasing to the infinity.

As non-T-prime graphs, we took (again) ring graphs of sizes $4, 6, 8, 10, 12$ and for each ring we repeated our algorithm 100 times per graph. In contrast to the earlier experiments, we were not interested in the precise result of each output now — we only counted the occurrences of success and failure.

With the following experiment, we provide a comparative context for the previous one. Having a ring of size $2n$ ($n \in \mathbb{N}$) the nodes of which are sequentially numbered in $0, \ldots, 2n-1$, we can construct a $n \times 2$-*grid* from that ring by inserting edges between node $(0+1) mod\ 2n$ and node $(0-2) mod\ 2n$, between node $(0+2) mod\ 2n$ and node $(0-3) mod\ 2n$, and so forth until there is no edge more to add. According to the previous experiment, we performed 100 repetitions for each grid of the sizes 2×2, 3×2, 4×2, 5×2, 6×2, (which are non-T-prime graphs as well) and we counted success and failure again. The results are shown in Appendix, second table. Compared to the previous experiment, we have added several new possibilities of communication here, namely those edges which transform the rings into grids, but obviously without a significant effect to the distribution of success and failure. Also in this experiment, the average success rate seems to be restricted to $\approx 80\%$, perhaps even slightly worse that in the previous experiment.[4]

[4] Similar contra-intuitive results are known from the area of

Also in this case, the presumed increase of success has not turned out to be very fast in terms of the increasing number of nodes in the graph. In **Fig.5**, a comparison between the results of this experiment and the previous one is sketched (see Appendix).

Acknowledgments

Thanks to *Afif Sellami* who has performed several of our experiments.

References

[1] G. Antonoiu and P. Srimani: *Distributed Self-Stabilizing Algorithm for Minimum Spanning Tree Construction.* LNCS Vol.**1300**, pp.480-487, 1997

[2] M. Bauderon, S. Gruner and M. Mosbah: *A New Tool for the Simulation and Visualization of Distributed Algorithms.* In B. Chaib-draa and P. Enjalbert, editors, *MFI'01, Toulouse, France*, volume I, pages 165–177, May 21-23, 2001.

[3] M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001. ENTCS, Volume 50, No 3.

[4] M. Bauderon, Y. Métivier, M. Mosbah and A. Sellami: From local computations to asynchronous message passing systems. LaBRI Report, Université Bordeaux 1, 2001. In preparation.

[5] M. Bauderon, Y. Métivier, M. Mosbah and A. Sellami. Graph relabelling systems : a tool for encoding, proving, studying and visualizing distributed algorithms. In *Getgrats Workshop, Bordeaux, France*, June 22-23, 2001. ENTCS, Volume 51.

[6] J.-E Burns and J. Pachl: *Uniform Self-Stabilizing Rings.* ACM Transactions on Programming Languages and Systems, Vol.**11** No.2, pp.330-344, 1989

traffic flow simulation, where the insertion of new roads causes even worse traffic jams instead of improving the situation.

[7] S. Dolev, A. Israeli and S. Moran: *Uniform Dynamic Self-Stabilizing Leader Election.* LNCS Vol.**579**, pp.167-180, 1991

[8] R. G. Gallager, P. A. Humblet, and P. M. Spira: *A Distributed Algorithm for Minimum-Weight Spanning Trees.* ACM Transactions on Programming Languages and Systems, Vol.**5**, pp.67-77, 1983

[9] Juan Garay , Shay Kutten and David Peleg: *A Sub-Linear Time Distributed Algorithm for Minimum Weight Spanning Trees.* SIAM Journal on Computing Vol.**27** No.1, pp.302-316, 1998

[10] I. Lavallée and G. Roucairol: *A fully distributed (minimal) spanning tree algorithm.* Inform. Proc. Letters Vol.**23**, pp.55-62, 1986

[11] A. Mazurkiewicz, *Solvability of asynchronous ranking problem.* Inform. Proc. Letters Vol.**28**, pp.221-224, 1988

[12] Y. Métivier, N. Saheb and A. Zemmari: *Randomized rendezvous.* In *Mathematics and computer science : Algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194, Birkhauser, September 2000.

[13] Y. Métivier and P.-A. Wacrenier: *A Distributed Algorithm for Computing a Spanning Tree in Anonymous T-prime Graphs.* LaBRI Technical Report No.1229-00, Université Bordeaux I, 2000.

[14] Y. Métivier, A. Muscholl and P.-A. Wacrenier: *About the local detection of termination of local computations in graphs.* International colloquium on structural information and communication complexity. Carleton University Press, pp.188-200, 1997.

[15] R. Pan, J. Wang and L. Chow: *A Self-Stabilizing Distributed Spanning Tree Construction Algorithm with a Distributed Demon.* Tamsui Oxford Journal of Mathematical Sciences Vol.**15**, pp.23-32, 1999.

[16] M. Raynal: *Distributed algorithms and protocols.* Wiley, 1988.

[17] Shing-Tsaan Huang: *Leader election in uniform ring.* ACM Transactions on Programming Languages and Systems, Vol.**155** No.3, pp.563-573, 1993.

[18] G. Tel: *Introduction to distributed algorithms.* Cambridge University Press, 2000.

A Figures

Ring Size	3	5	7	11	13	17	19	23	29
Messages	19, 15, 19, 19, 19.	52, 44, 31, 42, 40.	75, 90, 74, 65, 69.	126, 113, 140, 116, 132.	165, 156, 171, 138, 166	229, 212, 213, 233, 251	196, 260, 214, 259, 238	293, 294, 293 273, 312	381, 368, 458, 416, 418.
Average	~18	~42	~73	~125	~159	~228	~233	~293	~408

Figure 2: *messages in T-prime rings*

Clique Size	2	3	5	7	11	13	17	19	23	29
Edges	1	3	10	21	55	78	136	171	253	406
Messages	5, 5, 5, 5, 11.	14, 21, 15, 17, 12.	64, 34, 37, 61, 60	177, 91, 133, 89, 101.	337, 331, 457, 376, 348.	504, 540, 629, 581, 594.	973, 1215, 1720, 1258, 1409.	2403, 1425, 1829, 1960, 1708.	2403, 3637, 3064, 2858, 2557.	3869, 4037, 3131, 4363, 2879.
Average	~6	~16	~51	~118	~370	~570	~1315	~1720	~2904	~3659

Figure 3: *messages in T-prime cliques*

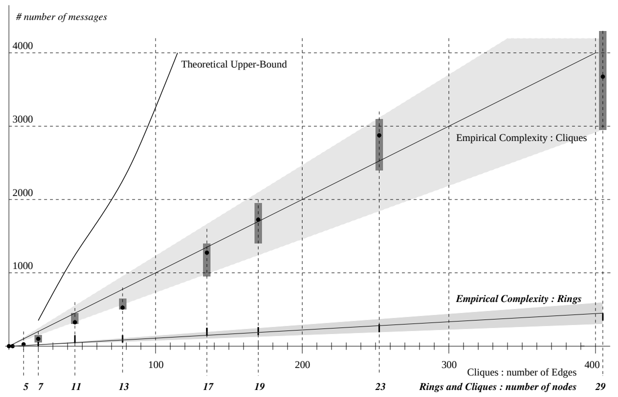

Figure 4: *complexity of SpanningTree on T-prime rings and cliques*

Algorithm on Non-T-Prime Rings

Ring-4: 82 success, 18 failure.

Ring-6: 72 success, 28 failure.

Ring-8: 86 success, 14 failure.

Ring-10: 77 success, 33 failure.

Ring-12: 84 success, 16 failure.

paragraph*Algorithm on Non-T-Prime Grids*

Grid 2×2: success rate 82%, see previous experiment.[5]

Grid 3×2: 75 success, 25 failure.

Grid 4×2: 83 success, 17 failure.

Grid 5×2: 79 success, 21 failure.

Grid 6×2: 79 success, 21 failure.

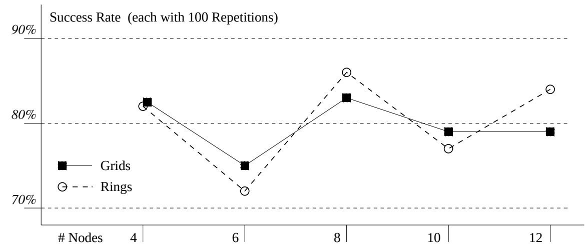

Figure 5: *comparison of SpanningTree success in rings and grids*

[5] As the 2×2-grid is the same object as a 4-ring, we take the result of the previous experiment in this special case for convenience.