Leader election on finite labelled graph with finite memory and anonymous agents

Nicolas Bacquey[1] and Gaétan Richard[2]

[1] INRIA Lille, Université de Lille, CRIStAL,
Villeneuve d'Ascq, France
nicolas.bacquey@inria.fr
[2] Université de Caen Normandie, UNICAEN, ENSICAEN, CNRS, GREYC,
Caen, France
gaetan.richard@unicaen.fr

Abstract. Given a finite graph of bounded arity on which lie several active "agents", the *leader election problem* consists in selecting exactly one leader among those agents. In this paper, we consider the case where it is possible to write a finite information on each cell and where anonymous agents have a finite memory and a local behaviour. In this case, we give an algorithm that can achieve leader election in every case where it is possible.

Introduction

Leader election is one of the main problems in distributed computing [1,2]. One of the origin of such a problem has to do with recreating a lost token on a token ring [3]. In this context, processors endowed with a unique ID on a regular ring must elect one leader. The main parameters are time and size of messages needed to achieve election. As this problem is very generic and can be easily formalised, it has been heavily studied in different contexts (e.g. asynchronous vs synchronous) either reflecting "real life" situations or theoretical cases [4,5].

Here, we focus on the *anonymous* case where processors have no unique identifier and thus are indistinguishable from each other [6]. In this case, they must also extract information from the underlying structure. Thus, an interesting parameter consists on information given *a priori* on the underlying structure and its influence. In particular, knowing the topology, the exact size of the network, or having an upper bound on the latter, change the solvability of the leader election problem (or variations such as tree construction problem) [7]. In this paper, we minimize the information given and we only assume that an upper bound of the degree of the graph is known. In this context, leader election is not always possible but limitations can be characterised with respect to symmetries of the underlying graph formalised by the notion of *fibration* [8].

Rather than assuming that every node is an active processor, a further restriction is the use of *agents* where processors are replaced by a fewer active and mobile elements [9]. The main difference with the previous case is that agents are

the only active elements. In this context, many algorithms have been developed to solve variety of problems depending on conditions and goals: for example, one can want to achieve *rendez-vous* when the graph is a tree [10] or with exactly 2 agents [11]. The case of *exploration* has also a received a lot attention [12]. In most of these works, one objective is to minimize time or size of memory used by the agent with respect to the size of the graph (in the feasible cases).

An alternative to the previous restriction is to consider the distributed model of *cellular automata* that consists on having finite-memory processors working synchronously on a regular grid. For them, leader election was studied by C. Nichitiu *et al.* in the case of finite portion of the regular grid by using the shape of the portion to achieve leader election [13,14,15]. Recently, N. Bacquey has proposed a new algorithm for leader election on periodic two-dimensional cellular automata [16]. This paper aims to extend this algorithm to the generic graph case.

To combine the two above restrictions, we consider in this paper agents endowed with **finite memory** and we allow them to read and write finite information on every vertex of the graph. In some way, this restriction can be seen as using Turing heads on the graph. Moreover, we only consider the **synchronous** case.

1 Context

In this paper, the model used is based on the model used in anonymous agent setting and in particular the one in [9]. The model is simplified to the synchronous case. For agents, the formalisation is derived from Turing head with additional neighbourhood looking capacity.

1.1 Automata on graphs

In the rest of the paper, the underlying structure is a finite undirected multigraph of bounded arity d as in the example given in Fig. 1. This graph is enriched by associating, for any vertex, a *port* to each outgoing edge ensuring determinism. Without loss of generality, we take those labels among the set $P = \{1, \dots, d\}$. In order to give a formal definition, it is easier to consider the labels to be put on the edges.

Definition 1. *A* finite labelled undirected d-graph *is a pair* $\mathfrak{G} = (V, E)$ *where* V *is a finite set of* vertexes *and* $E \subset V \times P \times P \times V$ *is the set of* labelled edges *satisfying that:*

- $P = \{1, \dots, d\}$ *is the set of* ports*;*
- *for any* $(v, p, p', v') \in E$, $(v', p', p, v) \in E$*;*
- *for any* $v \in V$*, the set* $\{p \mid \exists v', p', (v, p, p', v') \in E\}$ *is a subset of* P *without repetition.*

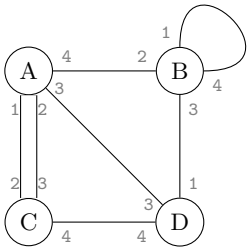


Fig. 1. Example of finite labelled undirected 4-graph

As some edges may not exist, some functions may be partially defined. To have a more uniform presentation, we introduce a symbol $\perp$ for undefined cases and denote $V^\perp = (V \cup \{\perp\})$.

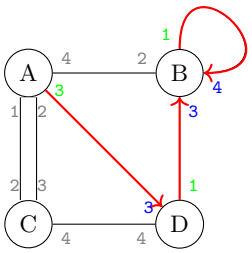
From such a graph, we define a *move* function (see Fig. 2), $\delta : V \times P \to V^\perp$ defined as $\forall v \in V, p \in P$,

$$\delta(v,p) = \begin{cases} v' & \text{if there exists } p' \in P \text{ such that } (v,p,p',v') \in E \\ \perp & \text{otherwise} \end{cases}$$

We denote the *return path* as

$$\mu(v,p) = \begin{cases} p' & \text{if } (v,p,p',\delta(v,p)) \in E \\ \perp & \text{if } \delta(v,p) = \perp \end{cases}$$

This function is extended for any argument in $V^\perp$ by $\delta(\perp,p) = \perp$ for any $p \in P$ and recursively any *path* $\boldsymbol{p} = p_0, p_1, \ldots, p_k \in P^*$ by $\delta(v,(p_0,p_1,\ldots,p_k)) = \delta(\delta(v,p_0),(p_1,\ldots,p_k))$. In the same way, we can extend the *return path* $\mu : V \times P^* \to P^{*\perp}$ by $\mu(v,(p_0,p_1,\ldots,p_k)) = \mu(\delta(v,p_0),(p_1,\ldots,p_k))\mu(v,p_0)$ when this is defined (that is $\delta(v,\boldsymbol{p}) \neq \perp$) and $\perp$ otherwise.



$\delta(A,3) = D$
$\delta(A,311) = B$
$\mu(A,311) = 433$
$\delta(B,433) = A$

$\delta(C,42) = \perp$
$\mu(C,42) = \perp$

Fig. 2. Example of moves and return paths

The graph is *connected* if for any pair of vertexes $v, v' \in V$ there exists a path $\boldsymbol{p} \in P^*$ such that $\delta(v, \boldsymbol{p}) = v'$.

As we use only this structure in our paper, we shall refer to it as simply graph:

Definition 2 (Graph).
In this paper, a graph $\mathfrak{G} = (V, E)$ *is a connected finite labelled undirected d-graph. The* size *of a graph is denoted as* $N = |V|$.

Over the vertexes of a fixed graph, we want to add some very simple undistinguishable "agents", automata with a finite memory, acting together. They are able to read and write colours, and to move, akin Turing machine's heads. This is formally done by adding two layers to the graph evolving along the computation: The first one is a colouring of the vertex over a finite set; the second one consists of automata described by their states. The absence of an automata being depicted with $\perp$. The resulting element is called *configuration* as depicted in Fig. 3.

Definition 3 (Configuration). *For a graph* $\mathfrak{G} = (V, E)$*, a* configuration *is a pair* $(\boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$ *with:* $\boldsymbol{\mathcal{C}} : V \to C$ *is a* colouring *of the graph with a finite set of colours* C*, and* $\boldsymbol{\mathcal{S}} : V \to S^{\perp}$ *where* S *is the finite set of* states.

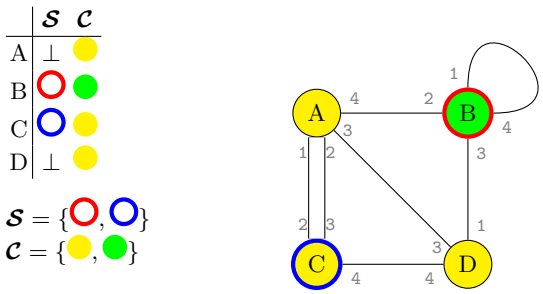
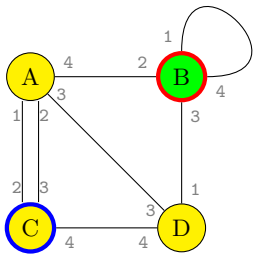


Fig. 3. Example of configuration

By convention, we extend $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{S}}$ by $\boldsymbol{\mathcal{C}}(\perp) = \perp = \boldsymbol{\mathcal{S}}(\perp)$.

As we want both our vertexes and automata to be anonymous (undistinguishable), this means that vertex labelling is hidden in our model and an automata has now way of saying whether or not a vertex reached by two different path is the same or not (except when using the return path). To depict this, we define the view of a vertex in a configuration as the set of information that an automaton can access. This is done by listing for any path, the colour, the state of automaton and the return path of the vertex reached by this path (see Fig. 4).

Definition 4 (View). *Given a graph* $\mathfrak{G} = (V, E)$ *and a configuration* $(\boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$, *the* view *of a vertex* $v \in V$ *is defined as* $\mathcal{T}(v) : P^* \to C^\perp \times S^\perp \times P^{*\perp}$ *by* $\mathcal{T}(v)(\boldsymbol{p}) = \boldsymbol{\mathcal{C}}(\delta(v, \boldsymbol{p})), \boldsymbol{\mathcal{S}}(\delta(v, \boldsymbol{p})), \mu(v, \boldsymbol{p})$.



$\boldsymbol{p}$	$\boldsymbol{\mathcal{C}}(\delta(v,\boldsymbol{p}))$	$\boldsymbol{\mathcal{S}}(\delta(v,\boldsymbol{p}))$	$\mu(v,\boldsymbol{p})$
ϵ	●	$\perp$	ϵ
1	●	○	2
2	●	○	3
3	●	$\perp$	3
4	●	○	2
11	$\perp$	$\perp$	$\perp$
12	●	$\perp$	12
...			

Fig. 4. Type of vertex A in the example

One first meaningful lemma is that this notion give birth to an equivalence notion [17,9]:

Lemma 1. *Given a graph* $\mathfrak{G} = (V, E)$ *and a configuration* $(\boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$, *having the same type is an equivalence relation whose classes have all the same size called* order *and denoted as* ν *(See Fig 5).*

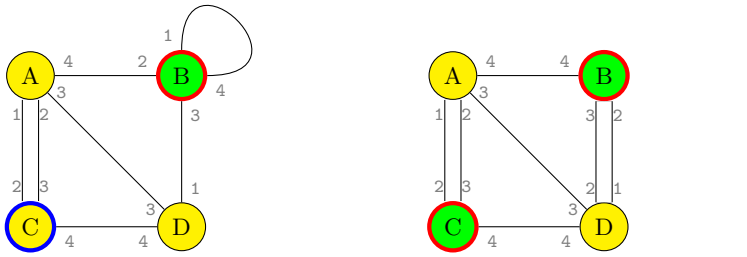
Proof. It is trivial that this notion is a equivalence relation.

Let $V = (v_k)_{0 \le k < n}$ and $V' = (v'_k)_{0 \le k < m}$ be two classes of equivalence. Without loss of generality, we can assume $n \ge m$. As the graph is connected, there exists a path $\boldsymbol{p} \in P^*$ such that $\delta(v_0, \boldsymbol{p}) = v'_0$ and a return path $\mu(v_0, \boldsymbol{p})$. As all elements in V have the same type, we have for any $0 \le k < n$, $\mu(v_k, \boldsymbol{p}) = \mu(v_0, \boldsymbol{p})$ and $\delta(v_k, \boldsymbol{p}) \in V'$. Since $\delta(\delta(v_k, i), \mu(v_0, i)) = v_k$, it follows that the function $v \to \delta(v, \boldsymbol{p})$ is an injection from V onto V'. This implies that $|V| = |V'|$. □

1.2 Dynamics

Dynamics of the system is based on several independent finite automata on a fixed graph. All automata follow the same local rule and are undistinguishable. In our model, the locality is understood as the adjacent vertexes to the position of the automaton (see Fig. 6). Formally, it can be defined as follows:

Definition 5 (Local view). *Given a graph* $\mathfrak{G} = (V, E)$ *and a configuration* $(\boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$ *and a vertex* $v \in V$ *such that* $\boldsymbol{\mathcal{S}}(v) \neq \perp$, *the* local view $\mathcal{L}(v) \in L = C \times S \times (C^\perp \times S^\perp \times P^\perp)^P$ *by* $\mathcal{L}(v) = (\boldsymbol{\mathcal{C}}(v), \boldsymbol{\mathcal{S}}(v), \{\boldsymbol{\mathcal{C}}(\delta(v,p)), \boldsymbol{\mathcal{S}}(\delta(v,p)), \mu(v,p)\}_{p \in P})$.



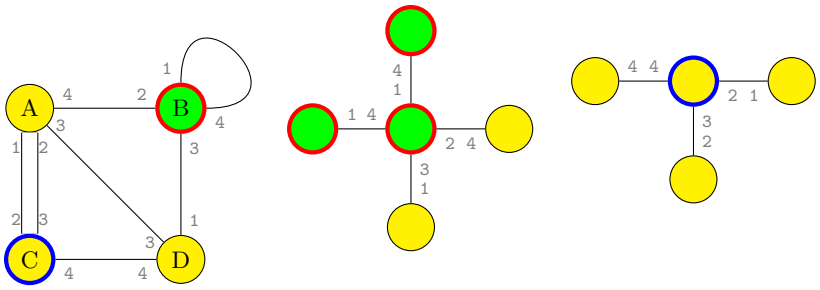
(a) A configuration of order 1 (b) A configuration of order 2

Fig. 5. Example of orders

One can remark that the local view corresponds to the restriction of the view of the vertex to paths of length at most one. As the only active places are those where an automaton is located, we restrict our definition to the vertex v where there is an automaton ($\boldsymbol{\mathcal{S}}(v) \neq \perp$).

With this information, the automata makes three actions: it changes its internal state, it changes the colour of its current vertex, and moves to a new adjacent vertex (or stays on the same vertex). All those actions are fully defined by the local view. In pseudo-algorithms used in this article, those actions are depicted by the words write, remove, or replace for colour interaction and move or stay for moves.

Definition 6 (Automaton rule). *An* automaton rule *is a total function* $F : L \to S \times C \times (P \cup \epsilon)$ *where* $L = C \times S \times (C^{\perp} \times S^{\perp} \times P^{\perp})^{P}$.



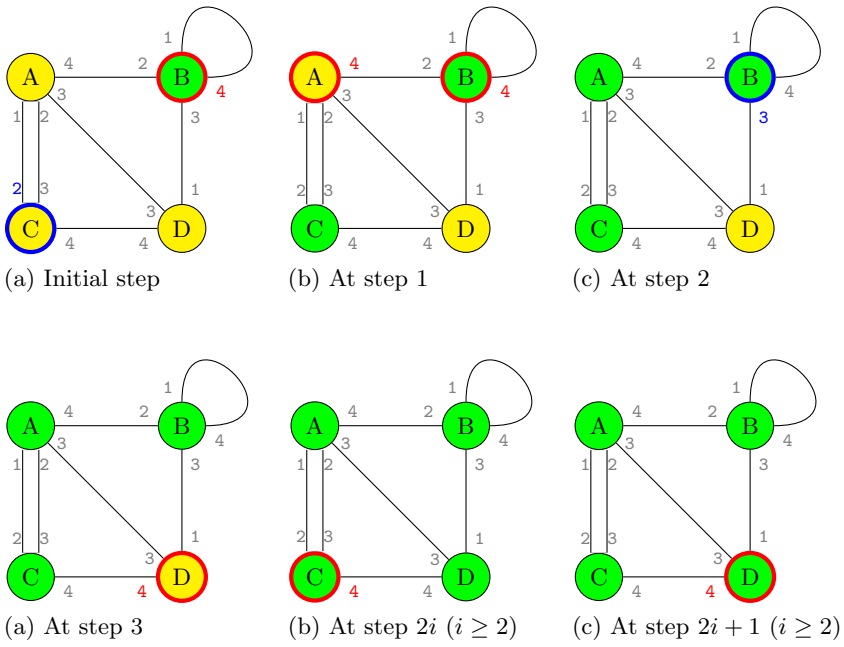
(a) The graph (b) Local view from B (c) Local view from C

Fig. 6. Example of local views

The global evolution of the system is done by applying synchronously the transformation of all automata on the graph as depicted in Fig. 7. Since there is

at most one automaton per vertex, the colour change does not pose any difficulty. The move, however, suffers from concurrency: what happens when two automata want to move into the same cell? This case is typical from concurrency and is very difficult to avoid in general: in our model, one can note that two automata can decide to move into the same vertex without seeing each other. In generality, even enlarging the local view does not help as automata cannot guess the move of each other.

There are several possible choices to deal with this situation: allow multiple automata per vertex endowing the result of the merge with a mutliset of all states, or give it a specific single state. In our case, we choose the later: if two or more automata move to the same vertex, the result is a unique automaton in a determined fixed *merge state* $s_m \in S$ (see step 2 of Fig. 7). This choice is the most generic one and our construction can be adapted for many different variations of merge.



(a) Initial step (b) At step 1 (c) At step 2

(a) At step 3 (b) At step $2i$ $(i \geq 2)$ (c) At step $2i+1$ $(i \geq 2)$

Fig. 7. Example of evolution.
There are two states: blue and red. In any case, the automaton paint its position in green. If the state is blue, it follows the least outgoing edge to a yellow node and turn red; Otherwise, it follows edge 4 . The merge state is blue.

Definition 7 (Image). *Given a graph $\mathfrak{G} = (V, E)$, a configuration $(\boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$, an automaton rule $F : L \to S \times C \times (P \cup \epsilon)$; the* image *$(\boldsymbol{\mathcal{C}}', \boldsymbol{\mathcal{S}}')$ is defined by:*

$$\boldsymbol{\mathcal{C}}'(v) = \begin{cases} c' & \text{if } \boldsymbol{\mathcal{S}}(v) \neq \bot \text{ and } F(\mathcal{L}(v)) = (c', \_, \_) \\ \boldsymbol{\mathcal{C}}(v) & \text{otherwise} \end{cases}$$

$$\boldsymbol{\mathcal{S}}'(v) = \begin{cases} s' & \text{if there exists a unique } u \in V, \text{ such that} \\ & \quad \boldsymbol{\mathcal{S}}(u) \neq \bot, F(\mathcal{L}(u)) = (\_, s', i), \text{ and } v = \delta(u, i) \\ s_m & \text{if there exists at least two } u \in V, \text{ such that} \\ & \quad \boldsymbol{\mathcal{S}}(u) \neq \bot, F(\mathcal{L}(u)) = (\_, \_, i), \text{ and } v = \delta(u, i) \\ \bot & \text{otherwise} \end{cases}$$

Since all the automata behave the same way, the system preserves some sort of "symmetries". This can be formalised using the view: if two vertexes have the same view in one configuration, then they also have the same view on its image. Since vertexes of different types can become identical, the size of each configuration class (the order) is increasing multiplicatively.

Lemma 2. *Given a graph $\mathfrak{G} = (V, E)$, a configuration $(\boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$ of order ν and its image $(\boldsymbol{\mathcal{C}}', \boldsymbol{\mathcal{S}}')$ of order ν' then ν' is a multiple of ν.*

Proof. The proof follows directly from the fact that the image only involves local views which are identical for vertexes having the same type. Thus the size of equivalence classes in the image is the sum of several sizes of equivalence classes of the configuration. As, by lemma 1, all those classes have the same size ν, the resulting size ν' is a multiple of ν. □

Those "symmetries" could also be characterised in terms of homomorphisms (often called *fibration* in this context) similarly as done in [18,19] but it goes beyond the scope of our paper.

1.3 Leader election problem

With this system, we can finally state our leader election problem. The intuitive idea is the following: given a graph and an unknown number of undistinguishable finite automata, can they work together to "explore" the graph and elect a leader among them without any prior or global information on the graph.

In a more formal way, we must define a starting configuration and some kind of halting condition. For the former, it is easy to do: we start with the graph coloured uniformly white and we put several automata on the graph all in the same initial state as in Fig. 8. More formally:

Definition 8. *Given a graph $\mathfrak{G} = (V, E)$, $c_0 \in C$ the* initial colour*, $s_0 \in S$ the* initial state *and $V_0 \subset V$ the* initial positions*, we define the* initial configuration *$(\boldsymbol{\mathcal{C}}_0, \boldsymbol{\mathcal{S}}_0)$ as:*

$$\boldsymbol{\mathcal{C}}_0(v) = c_0, \quad \forall v \in V, \boldsymbol{\mathcal{S}}_0(v) = \begin{cases} s_0 & \text{if } v \in V_0 \\ \bot & \text{otherwise} \end{cases}$$

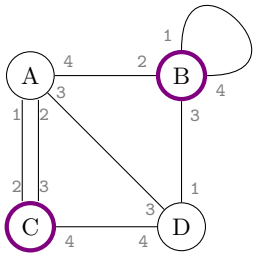


Fig. 8. Example of an initial configuration with 2 automata

One first easy remark is that, if there is no automaton, then the configuration does not change. So, we will always assume that $|V_0| \geq 1$. Another one is that the number of automata is decreasing during the evolution.

With this initial configuration, we can choose an automaton rule and let the dynamical system evolves.

To detect the end of the algorithm, one ideal method is to have each process enter a terminal state (*explicit termination*). Since our case is anonymous and do not have access to an upper bound on the size of the graph, it can be proven that no explicit termination algorithm exists (see Theorem 9.8 of [2]). Thus, we consider *implicit termination*: classically, it is defined as the step where no more messages are exchanged between agents. In our case, the strongest transposition would be that every agent does nothing and thus that the dynamical system enters a *fixpoint*. However, we shall only achieve a weaker notion of termination by considering when the system enters a *cycle* (which is always the case since we have a finite number of possible configuration).

Due to the uniformity of our system, there are several cases when it is not possible to distinguish between automata. This can be linked to the notion of view presented above: initially, it is easy to see that, on a graph $\mathfrak{G} = (V, E)$, an initial configuration $(\boldsymbol{\mathcal{C}}_0, \boldsymbol{\mathcal{S}}_0)$ has a number of automata multiple of its order ν. Moreover, this fact is preserved by evolution as stated in the following lemma:

Lemma 3. *Let $\mathfrak{G} = (V, E)$ be a graph and $(\boldsymbol{\mathcal{C}}_0, \boldsymbol{\mathcal{S}}_0)$ an initial configuration of order ν. The number of automata presents in the evolution of the configuration is a multiple of ν.*

Proof. Iterating lemma 2, we can deduce that, at any step, the order of a configuration is a multiple of the initial order ν. The number of automata being a sum of size of different classes, it is a multiple of the order and thus of the initial order. □

It can also be seen that this lower bound also applies to "marks" written on the graph. This allows us to give a formal (non-trivial) definition of leader election: we say that the election occurs if, after some time, the number of

automata reaches the lower bound. Recall that since our evolution only destroys automata, the limit exists and is stable once reached. Moreover, as we work on a finite case, this limit is reached in finite time.

Definition 9 (leader election). *A automaton rule F achieves* leader election *on the initial configuration $(\boldsymbol{\mathcal{C}}_0, \boldsymbol{\mathcal{S}}_0)$ of order ν on a graph $\mathfrak{G} = (V, E)$ if it eventually contains exactly ν automata.*

The main result of this paper is that we can exhibit a "universal rule", which given a fixed arity, works for any graph and any initial configuration.

The construction will be a little stronger regarding the halt and allows to "externally" detect the end of the process by the condition that *all* automata enter a specific subset of states.

Theorem 1. *For a fixed arity d, there exists an automaton rule F that achieves leader election on every initial configuration of any graph of arity d.*

The rest of the paper is devoted to prove the result. The intuitive idea of the algorithm is the following: each automaton is responsible for an exclusive portion of the graph making use of a spanning tree. The algorithm is divided in two steps.

The first step describes (in section 2) how each automaton make the expansion of its own exclusive portion over any non-claimed space. This step ensures that after some finite time the graph is partitioned into a forest of spanning trees where each tree has exactly one automaton.

The second step is a look and merge procedure: while always staying in its tree, every automaton looks at all its neighbours. The looking scheme is done so that if two adjacent trees are different, the corresponding automata will see each other and then merge together. This process is detailed in section 3.

All trees being identical is not sufficient condition to reach the minimal number of automata. In our case, we also need them to be regularly positioned. This problem is dealt with in section 4 by enriching the previous looking scheme to ensure merge when trees are not regularly positioned.

At last, we devote section 5 to bring all pieces together, study robustness of our algorithm to variation of definitions, and propose some easy or challenging possible perspectives for leader election.

2 Trees and Spanning forest construction

This section is devoted to present the first step of our leader election algorithm. Starting from an initial configuration, this algorithm construct a spanning tree forest partition where each tree has exactly one automaton on it.

First and foremost, the first action of any automaton in the initial state is to mark vertexes that initially contains automaton using a layer (called init_pos) on colour that will not be altered after this (Algorithm 1).

2.1 Tree encoding and successor

To encode the spanning rooted tree, we encode it by colouring the graph in the following way: a specific state ($\bullet$) encodes the root whereas each other node consists of an integer $f(v) \in P$ indicating the direction to its father and a list of its children (see Fig. 9).

During enlarging a tree or merging several trees, part of information stored is incoherent: it may happen that a father point towards a child but this child does not consider it as its father. For this case, we define the set of foster fathers of an vertex v as the list of port for which v is a child other than its father (formally: foster fathers$(v) = \{p | \exists p', v',\ (v, p, p', v') \in E,\ p' \in \underline{children}(v')$ and $p \neq \underline{father}(v)\}$

Along the algorithm, we will always ensure that there is exactly one automaton per tree.

Our algorithm heavily relies on the following notion of half-edge:

Definition 10 (half-egde).
A half edge *is a pair* $(v, p) \in V \times P$ *consisting of a vertex* v *and an port* p *.*

An half edge is *empty* if $\delta(v, p) = \perp$. During the algorithm, an agent can encode an half-edge by using only the (finite) value of a port, the vertex being implicitly define by its position. In algorithms, we refer to the current half-edge as CURRENT.

Definition 11 (mirror).
The MIRROR *of a non empty half-edge of* (v, p) *as the half-edge* $(\delta(v, p), \mu(v, p))$

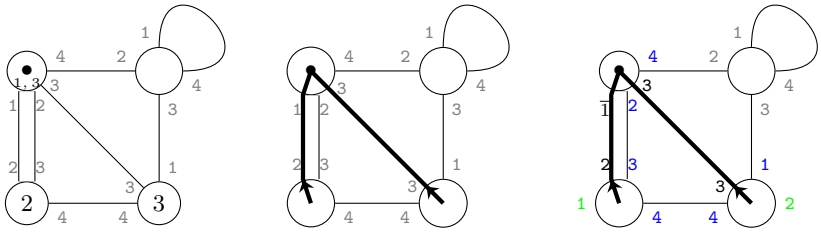
For a tree, we define its *size* n as the number its vertexes[3]. An half-edge is *internal* if it is part of an edge of the tree and *external* otherwise (see Fig. 9). Note that the mirror of an external half-edge may belong to the same tree. The *initial half-edge* is the edge $(r, 1)$ where r is the root vertex of the tree (this half-edge is not necessarily internal). At last, an half-edge (v, p) is *unclaimed* if either the vertex $\delta(v, p)$ or its mirror does not belong to a tree.

The basic brick of our algorithm is a traversal of all the half-edges inside the tree using a depth first search approach as depicted in Fig. 10 and formalised by the following definition of successor.

[3] The value of n is always smaller than N the size of the graph

Algorithm 1 Initialisation

```
1: procedure INIT
2:     write init_pos
3:     stay in place
4: end procedure
```



(a) Encoding of a tree (b) Visual representation (c) Half-edges
In the last figure, half-edges are depicted in black for internal, blue for external, dark blue for unclaimed and the initial one is overlined.

Fig. 9. Example of tree and half-edges classification

Definition 12 (successor). *The* SUCCESSOR *of an half-edge* (v, p) *is*

$$\begin{cases} (v, p+1 \mod d) & \text{if } (v,p) \text{ is external} \\ (\delta(v,p), \mu(v,p)+1 \mod d) & \text{if } (v,p) \text{ is internal} \end{cases}$$

Since it include all half-edge (even if the edge does not exists in the graph), the successor induces a cycle of length exactly $d \times n$.

2.2 Spanning Forest covering mechanism and merge

A covering of the whole graph by a forest is done in the following way: initially, each automaton (in the initial state) constructs a one-node tree limited to a root made of their current vertex.

Then, we introduce an algorithm that starting on the initial half-edge of an existing tree extends it maximally by running cycling thought successors incorporating any unclaimed half-edge it encounters.

This is done by going to the new vertex along with adding it as a children in the parent node (this creates a foster father on the new vertex). Then, the automaton marks the parent direction $f(v)$ in the new node and can continue the round from here (see Fig. 11). To be completely precise, the claiming does not occur if an automaton is present on the other side of the half-edge (since this automaton is in the process of claiming it).

The algorithm stops when it reaches the initial half-edge again.

One important point is that, since the round goes into the next half-edge only if it is claimed, the resulting tree at the end of Algo. 2 has no unclaimed half-edge.

A problem occurs in our algorithm when two (or more) automata claim the same vertex simultaneously without having seen each other. In this unavoidable case, the result is the following: we end with an automaton in the merge state

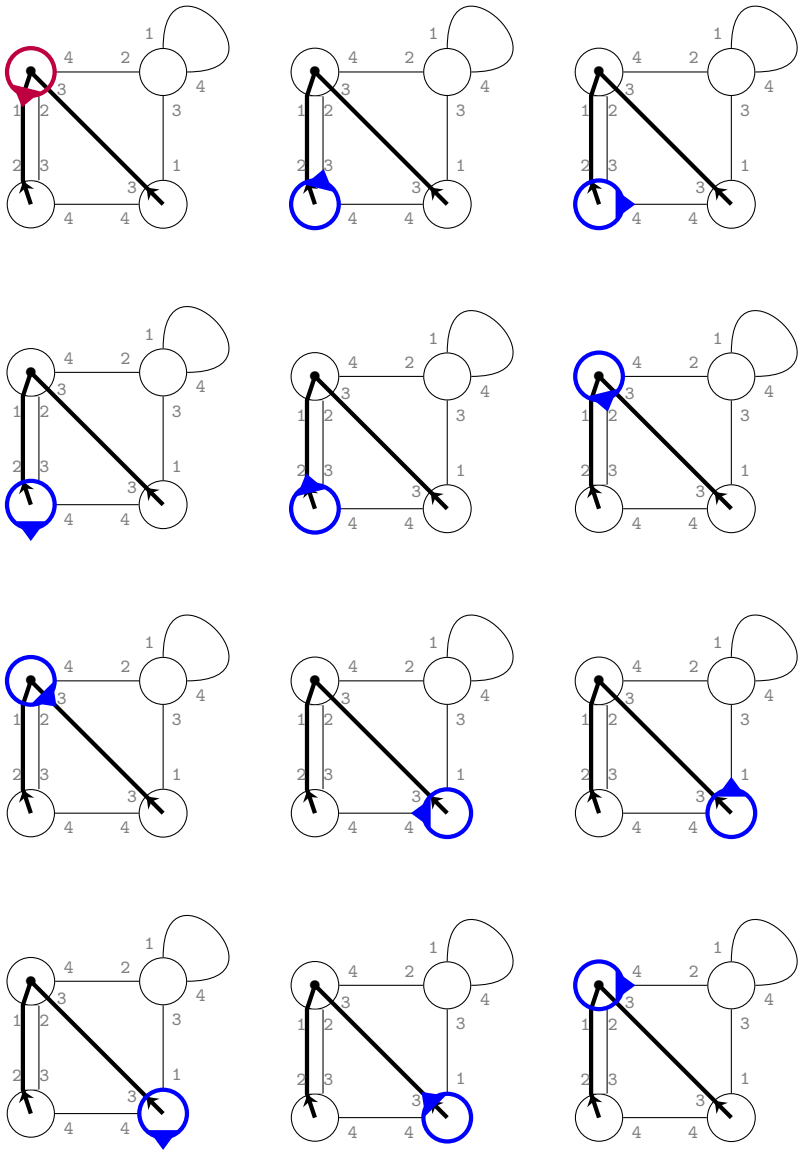


Fig. 10. Cycle of half-edges induced by successor relation in a tree with $d = 4$ and $n = 3$

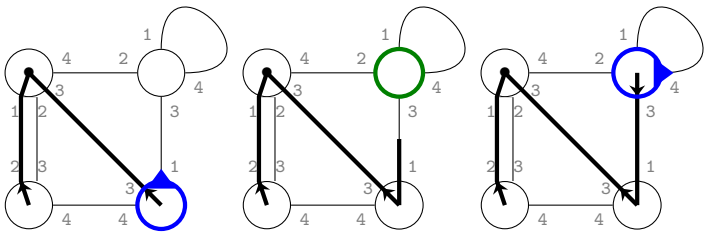


Fig. 11. An example of execution of Algo. 2 when MIRROR.vertex is not inside a tree

Algorithm 2 Spanning forest algorithm

```
1: procedure TREE
2:     repeat
3:         if MIRROR.vertex is not inside a tree then
4:             write (add) CURRENT.port to children
5:             move to MIRROR.vertex                          ▷ May lead to collision
6:             write foster father to father
7:             move to SUCCESSOR of (CURRENT.vertex, foster father)
8:         else
9:             remove all colours except father,children, and init_pos
10:                                                           ▷ Do nothing at the moment
11:            move to SUCCESSOR
12:        end if
13:    until CURRENT is intial    ▷ This condition is check before the previous move
14: end procedure
```

(over a vertex not belonging to any tree) that has at least two foster fathers, each being a distinct tree (that the automaton is responsible for).

Here, we choose to attach all other trees to the one accessible by the smallest index in the way depicted in Fig. 12. To do this, the automaton selects the largest foster fathers and goes back to the root of the automaton mirroring edges along the road; when it reaches the old root, it transforms it into a standard node and goes back to the vertex where the merge occurred (it can be done by going back in the tree until encountering a node without a parent). The automaton repeats the previous step until the merge vertex has only one foster father. In this last case, it simply transform the merge node it into a standard node and goes back to the root. At this point, it simply restarts the Algorithm 2. This can be written in Algorithm 3.

Algorithm 3 Merge trees (first version)

```
procedure MERGE
    repeat
        if —foster fathers— ¿ 1 then
            move to max(foster fathers)
            while CURRENT.vertex is not a root do
                replace father with foster father
                write (add) old father to children
                move to old father
            end while
        end if
        write foster fathers to father
        while father exists do
            move to father
        end while
    until CURRENT.vertex is root
    stay in place
end procedure
```

This algorithm achieves the desired result:

Proposition 1. *After a polynomial time, the application of Algorithms 2 and 3 starting from an initial configuration ends on every automaton and achieves a spanning forest.*

Proof. The first easy remark is that the size of any tree is always increasing. Locally, the automaton goes back to its father only if all adjacent half-vertexes are also in a tree. Thus, there is no unclaimed half-vertex adjacent to the tree once the algorithm finishes.

For the complexity, let us do a rough estimate: the maximal time for a tree to do a full walk when not interrupted by a merge is dn. If a merge happens, the tree grows by at least one and merge takes at most $2n$ steps. So there can be at most N merges leading to a $\mathcal{O}(N^2)$ bound on spanning forest construction. □

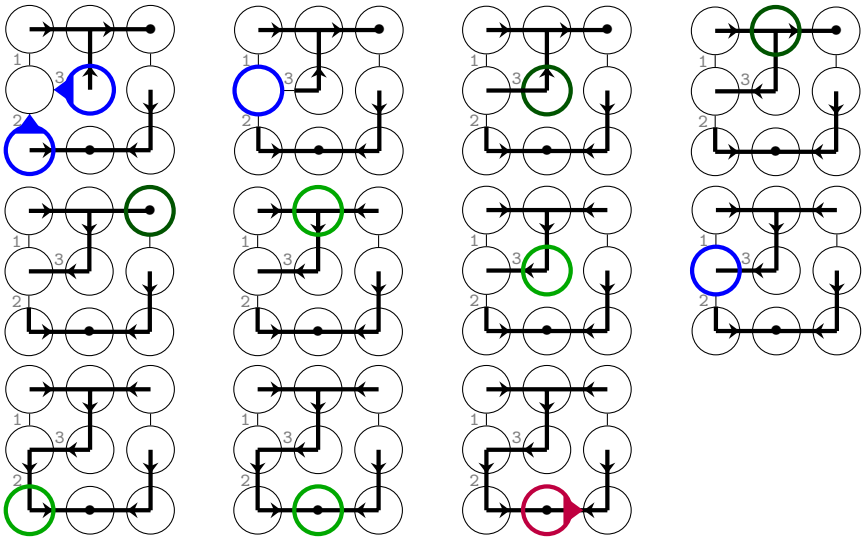


Fig. 12. Example when a merge occurs using Algo. 3

2.3 About merges

Before going in the next step of our algorithm, let us take some time to discuss about merge procedure. In fact, this merge is the only case where it happens on an uncontrolled way.

One first interesting remark is that our construction is very robust to variations in the choice of the merge process. In most other definitions (using a Cartesian product of state in the result, ...), the algorithm presented stays valid.

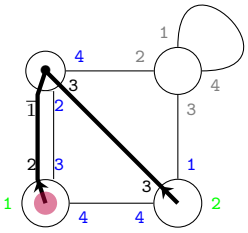
One other remark is that in our specific case this problem can be totally avoided by allowing the automata to have a local view of radius 2. In this case, any automaton willing to claim an empty vertex can see all other automata that wish to do so (and vice-versa) since the decision only involve the automaton and the state of the target vertex. Using the full order of the edge of the claimed vertex, we can ensure that exactly one automaton claims it.

2.4 Tree word

To characterise a tree, we associate with each tree a *Tree word* consisting in listing the half-edges encountered during a round (See Fig. 13). It is easy to see that the encoding is injective with respect to the tree and that the size of the word is dn. So this word encodes, in some sense, the local structural aspect of our configuration. To ease the work on the second layer, we prefix this word by a two letters prefix `#S`. The letter `S` will be used in section 4 to do one step of Algo. 10 and the letter `#` serves as a placeholder to guarantee concurrency.

We also add, using a Cartesian product over letters the fact that the vertex had an automaton in the starting configuration (see Fig. 13).

Definition 13 (Tree word).
The tree word *associated to a tree is the sequence of half-edges' indexes encountered during a round along with their status: internal or external, prefixed by* ***#S****.*



Tree word:
#S1341223412 34

Tree word with initial automaton input of Fig 8:
#S1341223412 34

Fig. 13. Example of tree word (see also the associated round in Fig 10)

One immediate point is to be able to use this word in our algorithm. Since we work with finite memory, we cannot store at one place the whole word. Thus, we design an algorithm that can extract on letter of the word at each call (writing it on the initial half-edge) and cycle when reaching the end of the word. The idea is to have a mark (called *tree_word mark*) on the half-edge corresponding to the current letter. Starting from the initial half-edge, the agent goes through all half-edges of the tree (using successors). When it encounters the tree_word mark, it store in its state the encoding of the letter and move the tree_word mark to the successor (see Algorithm 4).

To be complete, we must correctly deal with the # and S letters: we add a marker (S mark) only located on the initial edge; the case # is remembered by having no marker present. We can thus complete the algorithm as depicted in Algorithm 5. In a high view, this agent perform a full cycle of all half-edges.

Lemma 4. *Algorithm 5 allows to compute the encoded value of next tree word letter starting from the initial half-edge doing a full cycle.*

3 Merging trees by looking and waiting

After having constructed a spanning forest, the second part consists in merging trees until they are all identical. To avoid most of difficulties, the underlying idea is that each automaton is stuck to its tree and does not interact with other tree unless explicitly specified.

Algorithm 4 Compute next letter of tree word (simplified version)

```
1: procedure NEXT_TREE_LETTER
2:     value = None                                   ▷ Stored in the agent's state
3:     repeat
4:         if CURRENT has tree mark then              ▷ Current letter
5:             value = encoding(CURRENT)
6:             remove tree_word mark
7:             move to SUCCESSOR
8:             write tree_wordmark
9:         else
10:            move to SUCCESSOR
11:        end if
12:    until CURRENT is initial
13:    write value
14: end procedure
```

Algorithm 5 Compute next letter of tree word

```
1: procedure NEXT_TREE_LETTER
2:     value = None                                   ▷ Stored in the agent's state
3:     repeat
4:         if CURRENT is initial and has S mark then  ▷ S case
5:             value = encoding(S)
6:             replace S mark with tree_word mark
7:             move to SUCCESSOR
8:         else if CURRENT has tree mark then         ▷ Current letter
9:             value = encoding(CURRENT)
10:            remove tree_word mark
11:            move to SUCCESSOR
12:            if CURRENT is not initial then      ▷ Done unless it is the last letter
13:                write tree_word mark
14:            end if
15:        else
16:            move to SUCCESSOR
17:        end if
18:    until CURRENT is initial
19:    if value is None then                          ▷ # case
20:        value = encoding(None)
21:        write S mark
22:    end if
23:    write value
24: end procedure
```

The algorithm main idea is the following: each automaton will either be *looking* or *waiting* on an half-edge. When, a looking automaton see a waiting one on the mirror half-edge, they both merge their trees. To ensure that this happens, the key point is to exploit synchronism alternate waits and looks in such a way that "different" adjacent pair of trees, one will be looking on a half-edge and see a waiting mark of the other on the mirror half-edge.

This section is divided in three subsections: first, we give the high level description of the main alternation algorithm in section 3.1. Then, we give the necessary detailed implementation of the algorithm in section 3.2. At last, we prove in section 3.3 that this merging procedure will reduce eventually the number of trees until they are all "identical".

This section gives the algorithmic core of our construction but is not sufficient: in fact, it depends on an encoding of a tree (but not its neighbourhood). To ensure that having "identical" trees means that leader election is achieved, this sequence must be enriched with some additional elements. This is done in section 4.

3.1 Alternating looks and waits

Here, we shall describe the algorithm alternating between looks and waits phases. For the rest of the subsection, we consider a tree of *size* n (number of vertexes).

During the algorithm, the agent will either be considered *looking* or *waiting* on an half-edge. For such a tree, waits will have a quadratic time in its size. As this value will be a key point, we introduce it as follows:

Definition 14 (Waiting time).
The waiting time *is defined as* $\tau(n) = Kdn(dn+1)$.

With this tool, we can describe our main base synchronisation algorithm: the idea is to wait on every half-edge along with doing a full round looking between waits. As such, we can ensure that despite the wait, the automaton is regularly looking at every half-edge. After waiting on all half-edges, we just repeat a variable number of time looking rounds (according to some letter in the characteristic word computed in an initial looking cycle) in order to add a slight but sufficient desynchronisation for the case of trees with the same size. The algorithm also contains an initial looking round that will be used latter to compute the integer associated with letter of the tree word one example of such a cycle is given in Figure 18.

Lemma 5 (Look and wait high-level algorithm).

`0` *Do an initial looking cycle (performing* Next_Tree_Letter*);*
`1` *Wait* $\tau(n)$ *on the current half-edge;*
`2` *Go to the next half-edge;*
`3` *Do a looking cycle;*
`4` *Repeat from step* `1` *until reaching the initial half-edge;*
`5` *Do* w_i *looking cycles.*

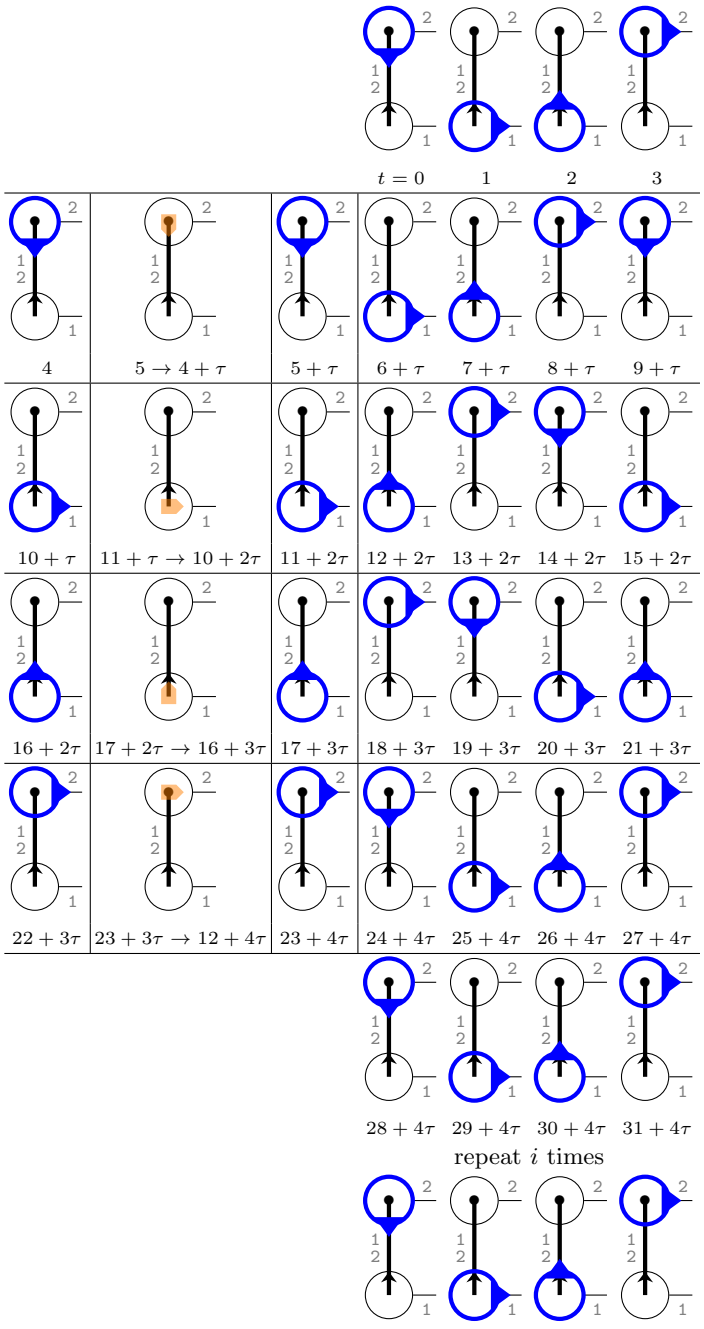
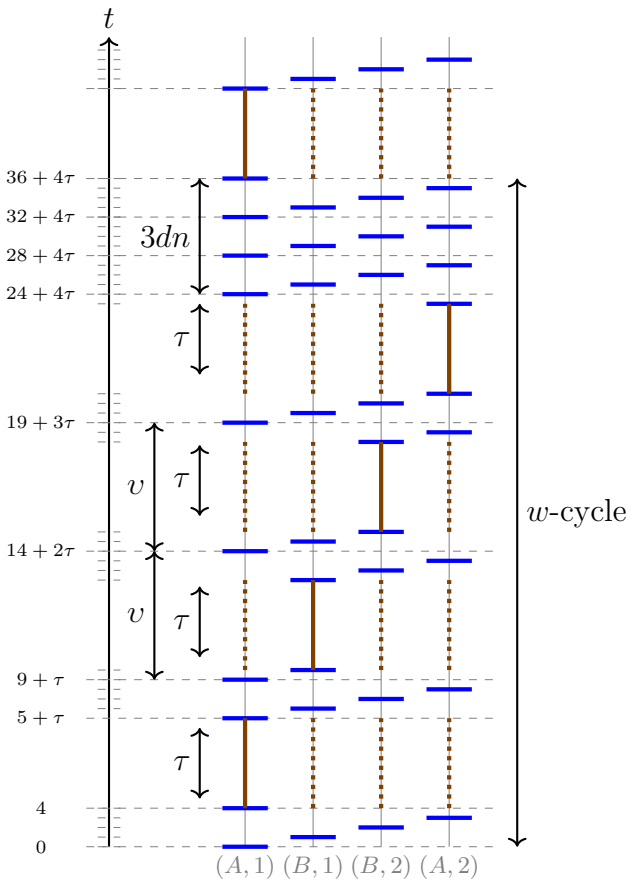


Fig. 14. Example of a cycle run (upper node is A, bottom one is B).

To easy understanding of the proof, we introduce in Fig. 15 a representation of the evolution of looking at a fixed half-edge.



Blue bars mark the time the automaton is present in the look state, plain dark orange line indicates the waiting mark is on this half-edge and dotted dark orange line times a waiting mark is on another half-edge.

Fig. 15. Symbolic representation of the cycle doing 3 additional round depicted in Figure 14.

In the following, we will mainly base our proofs on the two following facts about the lock and wait algorithm.

Lemma 6. *Algorithm* Look and wait *depicted in Lemma 5 ensures that:*

- *the waiting time is* $\tau(n) = Kdn(dn+1)$*;*
- *on any half-edge, the maximal time without a looking automaton is* $\upsilon(n) = \tau(n) + dn$*;*

Proof. The first point is evident by lemma 7.

For the second point, we change the focus from the automaton to the half-edge as done in Fig. 15. The longest time between two consecutive looking pass happens when the automaton has waited on another half-edge. In this case, the automaton has waited $\tau(n)$ steps, looked one time at any other half-edges $(dn-1)$, and looked one additional time on the waited half-edge. Thus we have $WR(n) = \tau(n) + dn - 1 + 1$.

If at some point, a looking automaton encounters a waiting mark on the mirror half-edge, both tree merge. As waiting and looking are exclusive, this ensures that exactly two distinct trees merge. Timing is chosen so that this encounter will eventually occur for different neighbouring trees (see section 3.3).

3.2 Implementation

The previous high-level vision hides several difficulties induced by our model. This section details implemetation.

Wait

One key element of our algorithm is that we design a method to wait for a quadratic time $O(n^2)$ (in the size of the current tree). However, due to the finite memory of our agent, it is impossible for it to stay more than a finite time on a vertex without looping indefinitely. The solution is to resort to active waiting where the agent put a *wait marker* and does some "useless" uninterrupted loop to count time. Thus, the wait algorithm is based on using the cycle of successors defined in Definition 12 that takes dn steps to complete. The core method to construct a quadratic time is to make a round each time we make a step of the initial round as depicted in Fig. 16 using as *step marker* (used as pebbles are in [20]).

Lemma 7. *Starting from looking at an half-edge, Algo. 6 does leave a mark for exactly $\tau(n)$ steps and then ends looking at the half-edge again.*

Proof. The behaviour is clear so we only need to look at the time spent waiting.

The inner loop is done exactly dn times and does one step. The outer loop thus takes $(dn+1)$ steps and is also done exactly dn times.

The whole algorithm is slowed down by a factor K. □

Merge

We shall now describe the merge procedure: this procedure is started when an automaton, looking at one half-edge, see a waiting marker on the mirror half-edge. As looking and waiting are mutually exclusive, this means that the marker belongs to another tree where an automaton is waiting. In this case, the looking

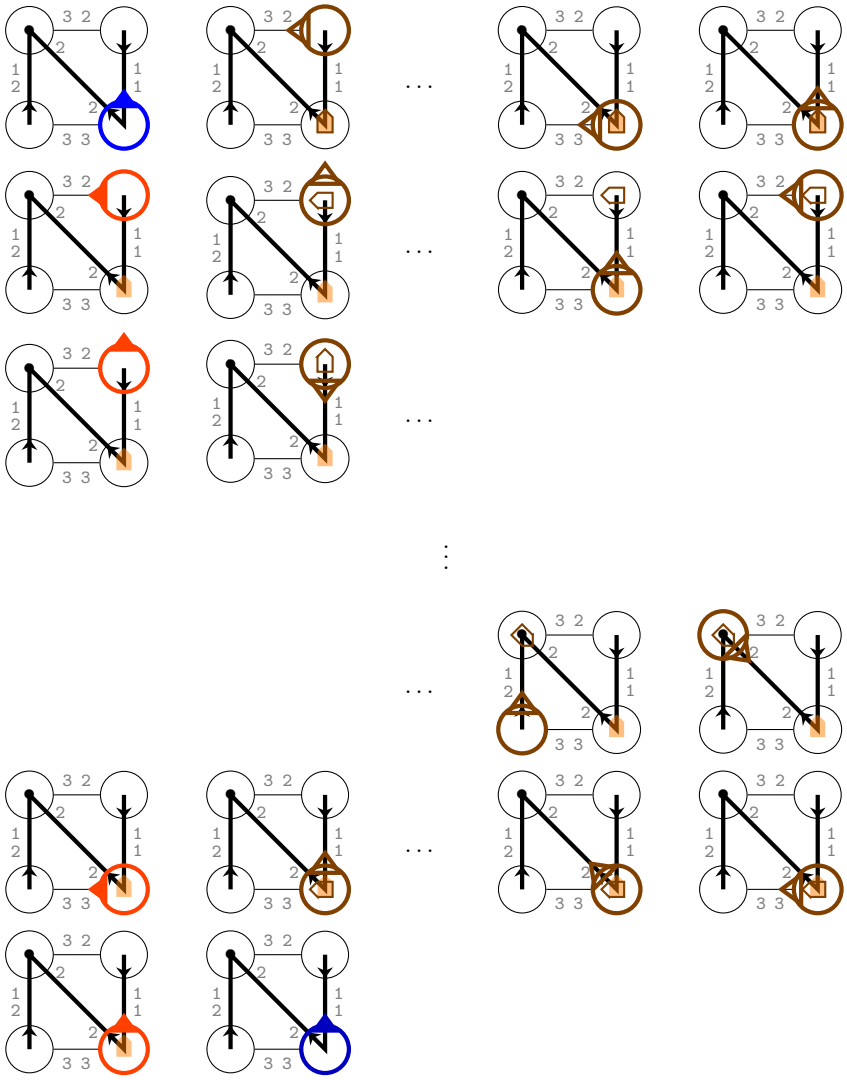


Fig. 16. Example of wait
The wait marker is depicted by symbol , the wait pebble by . During the round, the automaton is in state but can take the state to indicate it carries the wait pebble.

Algorithm 6 Wait

```
1: procedure WAIT
2:     write CURRENT wait marker .
3:     repeat
4:         write CURRENT step marker
5:         repeat
6:             move to SUCCESSOR
7:         until CURRENT has step marker
8:         remove step marker
9:         move to SUCCESSOR
10:    until CURRENT has wait marker
11:    remove wait marker
12:    stay in place
13: end procedure
```

automaton marks the parent direction and goes in the marker place . It stays there until the other automaton arrives and they both merge (see Algorithm 7 and Figure 17). By construction, we ensured that if there is a waiting mark, the automaton that is associated with this mark is ensured to be present in time linear relative to its tree size.

Algorithm 7 Look

```
1: procedure LOOK
2:     if MIRROR.vertex has wait marker then
3:         write (add) MIRROR.vertex to Children
4:         move to MIRROR.vertex
5:         loop                          ▷ loop doing nothing until a merge occurs
6:             stay in place
7:         end loop
8:     end if
9: end procedure
```

Once merged, the new automaton is in the same merge state as when the merge occurs in the spanning tree construction. However, it can know it is in a merge operation since the vertex, it is currently in, belongs to a tree (that is, the current state contains a father).

At this point, the automaton has just to go to the foster parent marker and to go back to the root of the tree inverting edges along the road (as was done in the first merge procedure Fig 12). Once done, the automaton is in a fully correct tree and can go back to the root of the tree. The former is already done in the first merge procedure and the latter is also already present. Thus, we only need to add an additional condition to the **if** statement and reuse the same procedure (see Algorithm 8).

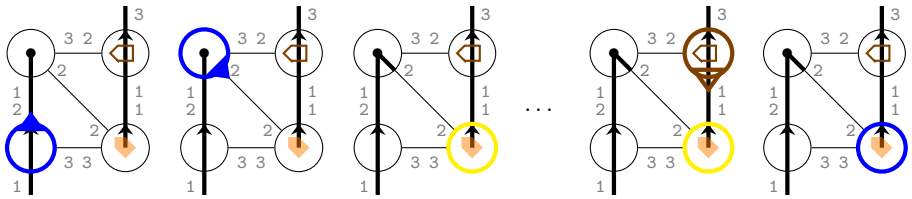


Fig. 17. Case when a merge occurs

Reusing the previous merge implies that once trees are merged, the agent will relaunch the spanning forest algorithm before restarting the (yet to define) main alternation. This is not a problem and even a good thing since we must clean all (non tree) information on the new merge tree and this was planned on line 9 of spanning forest algorithm (Algorithm 2).

Algorithm 8 Merge trees (second version)

```
procedure MERGE
    repeat
        if —foster fathers— ¿ 1 or ( father exists) then
            move to max(fosterfathers)
            while CURRENT.vertex is not a root do
                replace father with foster father
                write (add) old father to children
                move to old father
            end while
        end if
        write foster fathers to father
        while father exists do
            move to father
        end while
    until CURRENT.vertex is root
    stay in place
end procedure
```

Algorithm

The formal full definition of the algorithm is given in Algorithm 9 and depicted in details in Figure 18. This implementatiton is compatible with the high level description.

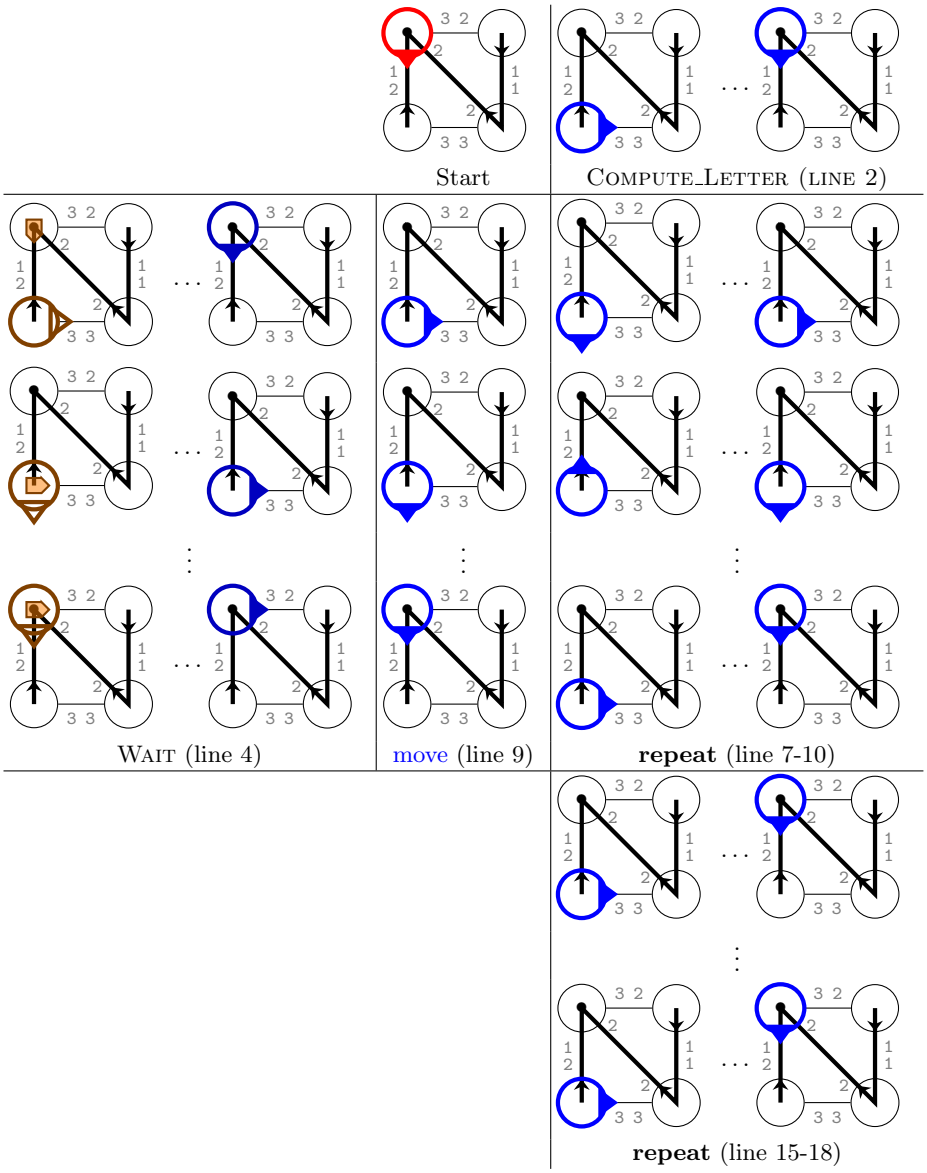


Fig. 18. Example of w_i-cycle

Algorithm 9 Cycle

```
1: procedure CYCLE
2:     COMPUTE_LETTER                          ▷ w_i is written on initial half-edge
3:     repeat
4:         WAIT
5:         move to SUCCESSOR
6:         write look peeble
7:         repeat
8:             LOOK
9:             move to SUCCESSOR
10:        until CURRENT is look peeble
11:        remove look peeble
12:    until CURRENT is initial
13:    repeat                                  ▷ (w_i times)
14:        replace (decrease) CURRENT counter  ▷ Current is initial
15:        repeat
16:            LOOK
17:            move to SUCCESSOR
18:        until CURRENT is initial
19:    until CURRENT counter is zero           ▷ Current is initial
20: end procedure
```

3.3 Prove of merges are occurring

Now that we have described our algorithm, let us see and prove it works. Core of the proof is to show that merges will occur until all automata are on identical trees. Here, we treat two distinct cases: whether sizes of the trees are different, or whether they have the same size but different characteristic sequences. The precise definition and construction of the characteristic sequence will be done in section 4.

Trees with different sizes merge

Proposition 2. *Given two automata A and B that are on adjacent trees of different sizes, then at least one of them will enter the merge procedure (in a polynomial time in the larger size).*

In this case, the intuitive idea is that the quadratic wait allows the larger one to wait sufficiently to ensure that the smaller one will be looking at it during the wait.

Proof. Let us assume that neither A, nor B merges with another (third) automaton; then we will prove that they will merge together in polynomial time.

Let us denote as n_a (resp. n_b) the size of A (resp. B) and assume that $n_a > n_b$. Since A and B are adjacent, there exists (at least) an edge between then.

Let us look at what happens during the time A is waiting on this half-edge using times given in lemma 6. On the one hand, the waiting mark for A will stay for $\tau(n_a) = Kd^2 n_a(n_a + 1)$. On the other hand, automaton B is ensured to pass looking on the opposite of this half-edge every $1 + \upsilon(n_b) = \tau(n_b) + d * n_b + 1$.

Then, we can show that:

$$\tau(n_a) = Kdn_a(dn_a + 1)$$

$$\tau(n_a) \geq Kd(n_b + 1)(dn_b + 2)$$

$$\tau(n_a) \geq Kd(n_b)(dn_b + 2) + Kd(dn_b + 2)$$

$$\tau(n_a) > Kd(n_b)(dn_b + 1) + Kd^2 n_b + 2Kd$$

$$\tau(n_a) > \tau(n_b) + dn_b + 2$$

$$\tau(n_a) > \upsilon(n_b) + 1$$

This implies that during the time the automaton A has waiting marker on the half-edge, the automaton B will be looking at this half-edge and thus initiate a merging.

Now, consider the time needed for a merge to appear: a rough estimate can be done as follows: the merge occurs during any w-cycle of A. Time for such a cycle is in $O(n_a^3)$ and it appears in look and wait algorithm after at most $O(n_a)$ (the additional rounds). Thus the former gives a polynomial bound on the occurrence of a merge. □

Trees with same size merge

PASS 2

If both trees have the same size, the idea is to use the characteristic sequence to differentiate them. As we do not have any information on when the wait and look algorithm is started on each tree, there may be a shift between the letters considered in each characteristic sequence. Thus, we consider the case when the sequences $c, c' \in C^*$ are *asymptotically different* — formally, for any $i, i' \in \mathbb{N}$, there exists $k \in \mathbb{N}$ such that $c_{i+k} \neq c_{i'+k}$.

The idea is to prove by contraposition by studying when trees do not merge.

Proposition 3. *Let A and B be two automata that are on adjacent trees of same size. If they do not enter the merge procedure, then their characteristic sequences are asymptotically identical.*

The rest of the section is devoted to prove this proposition. Let $n = n_a = n_b$ be the common size.

In this case, the key point is the number of additional rounds done at the end of a w-cycle. The idea is that they will introduce some small shift between the behaviour of the two automata leading to their merge.

First, let us show that there exists some kind of "synchronisation" between the two automata. Same as previously, let us look at what happens during the

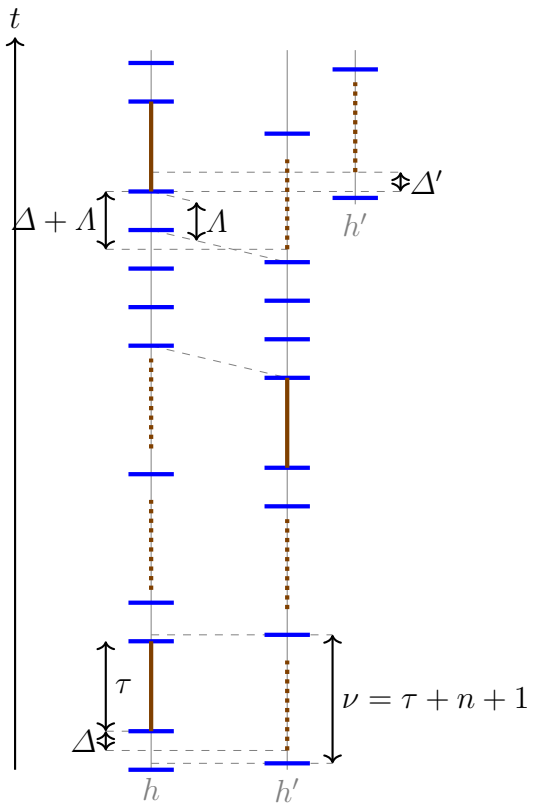


Fig. 19. A comparative vision of behaviours on two mirror half-edges h and h' during more than one w-cycle. The left h' representation use constraints at the beginning of a cycle and assume the two cycles are not identical; the right one use the same constraint at the beginning of the next cycle. Combining both ends in the contradiction explained in the proof. For concision, some values are not realistic (for example, we use $\Delta = n$.

time A is waiting on one fixed half-edge h adjacent to h' in B (see Fig. 19). Since both have the same size, we cannot hope that B will pass looking at this half-edge during the wait. However, if B does either wait or look, the fact that B does not look at this half-edge implies that it also does wait during part of this time.

Let us look at the (only) wait of B during the wait of A. Let us define as *desynchronisation* Δ the number of steps between the start of the wait for A and the one for B. Since before starting to wait, B does a full looking round and has not encountered A waiting, it implies that $\Delta < dn$. The same way, since B is doing a step then a full looking round after waiting, we can deduce that $-dn < \Delta$.

This notion will be the key to prove the following lemma which will also be reused latter.

Lemma 8. *Let A and B be two adjacent trees of the same size that do not merge with each other. When A is waiting on a half-edge adjacent to B during a w_a-cycle, B is also doing a w_b-cycle with $w_b = w_a$.*

Proof. Using the previous paragraph, when A is looking on an half-edge adjacent to B, then B is doing some w_b-cycle with a desynchronisation Δ.

By contradiction, let us assume that $w_a \neq w_b$ as in Fig. 19.

Let us look at what happens when A is looking at the same half-edge in the next cycle. Since letters are different, the lengths of the two cycles differ by Λ which is at least $3dn$ and by at most $3|C|dn$ (formally $3dn \leq |\Lambda| \leq 3|C|dn$). Thus, we can deduce that B starts a wait $\Delta + \Lambda$ after (or before) A starts its wait.

We can compute that $2dn \leq |\Delta| - |\Lambda| \leq |\Delta + \Lambda| \leq |\Delta| + |\Lambda| \leq (3|C| + 1)dn$. Since this value is not compatible with the desynchronisation, this implies another start of waiting for B exists Δ' after (or before) A starts to wait for B.

Those two starts of wait are separated by at most: $|\Delta + \Lambda| + |\Delta'|$. However:

$$|\Delta + \Lambda| + |\Delta'| \leq (3|C| + 1)dn + dn$$

$$|\Delta + \Lambda| + |\Delta'| \leq (3|C| + 2)dn$$

and

$$(3|C| + 2)dn < Kd^2n(n + 1) = \tau(n)$$

for sufficiently large K (since $n \geq 1$ and $d \geq 1$).

This would imply that both starts are refer to the same moment and thus $\Delta + \Lambda = \Delta'$. The fact that $|\Delta' - \Delta| < 2n < |\Lambda|$ achieves the contradiction. □

To prove proposition 3, we associate to any cycle done by A reading a letter an identical cycle done by B. Since this association is injective, they have the same characteristic sequence. Of course a corollary of the previous result is that automata with asymptotically different characteristic sequences merge.

4 Extending tree word to neighbours

PASS 2: problem

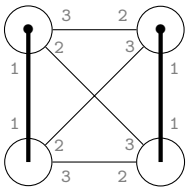


Fig. 20. Example of identical trees with a non regular placing: both trees have the tree word #S$\bar{1}$23123

In this part, the idea is to extend the sequence in order to get information about the path to the neighbour tree's root for each external half-edge.

The main problem about this piece of information is that it is outside the tree. Since keeping the automaton inside the tree is one of the essential property of our algorithm, we design a specific way of retrieving the path: each automaton is responsible to put this information for every of its external half-edge so that its neighbours can get this information from their respective trees.

The basic idea is to start by putting the index of the father on each vertex and then push the indexes from the root to the leaves until no more indexes are left (see Fig. 21).

Algorithm 10 Sending path to edge

Starting from a tree with ϵ on each vertex:

1 Put the index of its father on each vertex (ρ for the root);
2 push the index currently in the father to all of its children (and add an ϵ on the root);
- if there still exists a non ϵ symbol, repeat 2; otherwise restart from 1.

The algorithm is executed one step for each full repetition of the tree word. More precisely, one step is executed during the initial round of letter S.

Now, the idea is to read this data from the neighbour's tree and incorporate into our sequence. This is done by taking the value when looking at the letter of the tree word corresponding to any external edge as depicted in Fig. 22. As a special case, when the value is ρ, the automaton take the return path of the half-edge.

If we avoid a lot of potential concurrency problems by keeping every automaton to its tree, there still is a question of synchronisation during the transmission

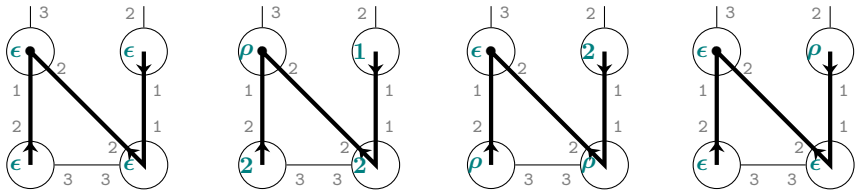


Fig. 21. Example of sending path to edge algorithm

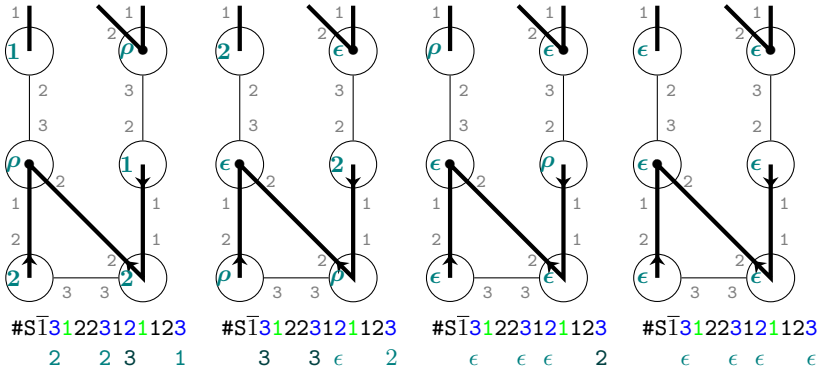


Fig. 22. Example of full characteristic sequence

of the path. For our case, we just need to prove that it works for trees of the same size, leading to the following proposition:

Proposition 4. *Let w be a characteristic sequence of a tree T of size n. Let T' be an tree of the same size n adjacent by a fixed half-edge, then the path from the half-edge to the root of T' is determined by the sequence of T.*

Proof. Notice that since we assume the existence of the sequence, we assume that no merge occurs.

Since it is trivial that the word sent by T' removing the last letter correspond to the path, the only point to prove is that the sequence read by T is effectively the one sent by T' and can be recovered from the infinite sequence.

To prove this, the first step is to show a synchronisation fact: between two consecutive reads by T, T' does exactly one write (a step of sending to path). For that, remember that each step of sending to path is done at the beginning of the `S`-cycle. By lemma 8, at some point during the `S`-cycle of T', T is doing also a `S`-cycle. Thus, we can deduce that this step occurs either during the `S`-cycle of T or during the previous `#`-cycle (since we can without loss of generality force that a `#`-cycle is longer than a `S`-cycle by choosing the encoding of `#` to be larger than the one of `S`). As all reads are done outside of those cycles, this conclude read letters are exactly the one present throughout the execution of the sending path algorithm.

The next step is now to extract the path from the infinite sequence. To do this, it is sufficient to notice that the path starts from the first non-ϵ letter (and ends with the last but one letter of this kind). □

5 Full algorithm, conclusion, and perspectives

5.1 The global algorithm and its proof

Let us combine all the previous elements to achieve the main result. The global algorithm consist just on doing a forest spanning algorithm followed by an (infinte) look and wait algorithm.

Algorithm 11 Leader election algorithm

The *leader election algorithm* is achieved by doing, starting from the initial state:

0 Initially mark the vertex (corresponding to positions of automata in the starting configuration);
1 Do a pass of spanning forest algorithm (Algo. 2); If in the merge state but not on a tree, execute merge I (Algo. **??**) and start this step again.
2 Repeat look and wait algorithm (Algo. 6) with characteristic sequence computation as depicted in section 4 and doing a merge II (Algo. 8).

To prove the good behaviour of our algorithm, let us take three steps: first, let us prove that the algorithm converges to a state where we have a covering of trees with the same characteristic sequence. To refine this result, let us show next that this convergence occurs in polynomial time. At last, let us prove that leader election is achieved.

Proposition 5. *Starting for any initial configuration on a d-graph, the leader election algorithm reaches a cycle where the graph is covered by trees with the same characteristic sequence.*

Proof. The first easy fact is that there is only a finite number of merges as it reduces the number of automata.

One can remark that an automaton in a merge state can distinguish whether it is in step 1 or in step 2 by looking whether there is a tree mark on the colouring of the current vertex and thus determine if it must use merge I or merge II.

The fact that trees achieve a coverage is directly derived from proposition 1 and is preserved by look and wait algorithm since it does never remove a vertex from a tree and does not interfere with the spanning forest algorithm as the automaton only exits its tree to go into a waiting half-edge which can only occur if the corresponding other automaton is also on the look and wait algorithm.

Once all automata are in the look and wait algorithm, a merge will occur if there is two neighbouring automata that either have different sizes (proposition 2) or have the same size but asymptotically different characteristic sequences

(proposition 3). Since the number of merge is bounded (and since our graph is connected), it implies that eventually all trees have asymptotically the same characteristic sequence. Since the common characteristic sequence is periodic, it immediately implies sequences are equals. □

As an important note, let us look at the efficiency of our algorithm. This analysis is far from optimal but we can still easily prove that our algorithm works in polynomial time.

Proposition 6. *The previous convergence is achieved in polynomial-time.*

Proof. Let us follow the previous proof method looking at time. Let n be the size of our graph (that is also a upper bound on tree sizes).

The number of merges is bounded by n.

The covering described by proposition 1 is bound by $O(n^3)$ (given in the proof).

For different size trees merging (proposition 2), the merge will occur during the time the automaton in the larger tree wait on the adjacent half-edge and so will occur before the end of the first w-cycle (thus in $O(n^3)$ time) and last $O(n)$ steps. As there is at most n merges, all trees will have the same size after at most $O(n(n^3 + n)) = O(n^4)$ steps.

For same size trees with different characteristic sequences, the merge will occur before the next cycle after the first different letter is encountered (see proof of proposition 3). The characteristic sequence is constructed by repeating the tree word (of size $O(n)$) changing at each repetition the second layer over the path word which is also of size $O(n)$. This implies that the characteristic sequence is of period $O(n^2)$ and thus that the merge occurs before $O(n^3 * n^2) = O(n^5)$ and still lasts $O(n)$. The already used limit of n merges implies no merge occurs after $O(n^6)$.

Combining all those bounds gives us a $O(n^6)$ bound. □

Last but no least, let us prove that when we have identical trees, leader election is achieved.

Proposition 7. *If the graph is covered by trees with the same characteristic sequence then leader election is achieved.*

Proof. By definition 9, achieving leader election means that the number of automata is equal to the order of the initial configuration. As there is exactly one automaton per tree and that the lower bound is enforce by definition, it is sufficient to prove an upper bound on the number of trees.

As the order is the cardinal of all equivalence classes over the type, it is sufficient to prove that the initial type of a vertex in a tree is fully determined by its position in the tree.

Recall (definition 4) that the type is a function over the possible path giving: the colour of the reached vertex, the state of any present automaton and the return path. As we consider the initial configuration, the colour is necessarily

the initial colour and the automaton is in the initial state. Thus, it is sufficient to check whether an automaton was here or not (which is recorded by a marker on the vertex in the current configuration) and the return path (for completeness, note that the return path gives the information on the existence of a vertex reached by the path).

Let us prove the latter using a recurrence over the length of the considered path.

For path of length 0 (that is for ϵ), the only needed piece of information (initial presence of an automaton) can be read in the current colour of the vertexes. As this data is used in the tree word, it is the same for all vertexes at the same position in a tree.

Let us now take a path $\boldsymbol{p} = p_0\boldsymbol{p'}$ with $p_0 \in P$ and distinguish according to the first half-edge taken by applying p_0. Since the trees are the same, the cases are the same for all vertexes in a same position.

- If the half-edge is empty then the result is $\perp$.
- If the half-edge is internal, as the trees are all the same, the return letter having taken p_0 is the same and the reached vertex has the same position for all vertexes at the same position in a tree. Applying the recurrence property allows to conclude.
- If the half-edge is external, then the two previous information can be recovered looking at the second layer of the characteristic sequence (see proposition 4), leading to the same conclusion as in the previous case.

□

From those three previous propositions, we can immediately deduce our main theorem 1.

5.2 Robustness and extensions

Order on the port

Let us now look at some more properties of the algorithm that do not directly appear in the current statement.

The first point is to look at the end mechanism: the algorithm ends in a loop. Moreover, each automaton can estimate the polynomial bound sufficient to ensure that all its immediate neighbours have the same characteristic word. They can enter thus a specific subset of states to have a global halting condition of all automata being on this subset of states. In the one dimensional similar case [21], one trick is used to erase all intermediate states used during the algorithm, to end in a fixpoint, and to recover the erased data when needed. Doing a recover phase in our case is an open problem.

Moreover, even if the algorithm depends on the arity d of the graph, this dependency is quite uniform. The depiction of the algorithm does not change. The key limitation is that the memory needed (size of the set of states) and the number of colours depends on the arity. Indeed, the automaton uses states to remember the current half-edge and thus need at least a memory linear in

the arity. The same is true to encode the tree in the colours. Moreover, for the latter, it seems difficult to restrict to a finite number of colour. However, one can note that the memory is linear in the arity and that most encoded data refer to half-edge. Thus, if we consider a model where colours and automata are put on half-edges, then we can obtain a universal rule which does not depend on arity. One can ask if this is also valid in the model where colours and automata are put on edges. For this more complicated case, the answer is not clear.

One other possible question is to look at possible variations in our graph structure used. The fact that the graph is undirected is an obvious necessity for our method to work. A more subtle question is the necessity of encoding the set of indexes as integers: most of the time, our algorithm only needs to go to the *next* index and thus, does only use the cyclic successor. Other than that, the minimum of the indexes is used only on very specific occasions: for the initial step (but this initial half-edge could be considered as part of the initial configuration) and when merges occur during the spanning forest algorithm. For the latter, one could expect that the method avoiding unexpected merges as depicted in section 2.3 could help but it also needs the minimum. For now, we do not know a mean to do without this minimum property.

One other major choice made concerns the initial configuration. We assume that both the underlying graph is in a clean state and the automata start synchronously. This suggests many possible extensions and questions. Concerning the synchronous start of automata, this is never used inside the algorithm. Taken alone, there is no problem in having delayed starts from automata.

Regarding states of the automata, we can remark that many data can be recovered from the colours. For example, knowing if the automata has finished Algo. 2 can be detected by knowing whether or not the associated tree has an unclaimed half-edge. From this point of view, this algorithm seems to be robust with regards to errors on the state of the automaton.

For colours, we could also take into account some set of initial colours (as long as it is a strict subset of the working set) by backing the colours as done for the initial position in a similar way done in [16]. If the set of initial colour can use any colour, the question becomes more tricky and is open. One possibility could be to look at similar questions and methods done in [22]. As in the previous paragraph, the question of robustness may be interesting but in the case of colours, the answer seems to be that our algorithm is very sensible to any slight error in the colours.

References

1. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
2. Tel, G.: Introduction to Distributed Algorithms. 2 edn. Cambridge University Press (2000)
3. Le Lann, G.: Distributed systems - towards a formal approach. In: IFIP Congress. (1977) 155–160

4. Korach, E., Kutten, S., Moran, S.: A modular technique for the design of efficient distributed leader finding algorithms. In: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing. PODC '85, New York, NY, USA, ACM (1985) 163–174
5. Chalopin, J., Métivier, Y.: Election and local computations on edges. In Walukiewicz, I., ed.: Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Volume 2987 of Lecture Notes in Computer Science., Springer (2004) 90–104
6. Yamashita, M., Kameda, T.: Electing a leader when processor identity numbers are not distinct (extended abstract). In: WDAG. Volume 392 of Lecture Notes in Computer Science., Springer (1989) 303–314
7. Yamashita, M., Kameda, T.: Computing on anonymous networks: Part i-characterizing the solvable cases. IEEE Trans. Parallel Distrib. Syst. **7**(1) (1996) 69–89
8. Boldi, P., Shammah, S., Vigna, S., Codenotti, B., Gemmell, P., Simon, J.: Symmetry breaking in anonymous networks: Characterizations. In: Fourth Israel Symposium on Theory of Computing and Systems, ISTCS 1996, Jerusalem, Israel, June 10-12, 1996, Proceedings, IEEE Computer Society (1996) 16–26
9. Dereniowski, D., Pelc, A.: Leader election for anonymous asynchronous agents in arbitrary networks. Distributed Computing **27**(1) (2014) 21–38
10. Fraigniaud, P., Gasieniec, L., Kowalski, D.R., Pelc, A.: Collective tree exploration. Networks **48**(3) (2006) 166–177
11. Dessmark, A., Fraigniaud, P., Kowalski, D.R., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica **46**(1) (2006) 69–96
12. Das, S., Flocchini, P., Kutten, S., Nayak, A., Santoro, N.: Map construction of unknown graphs by multiple agents. Theor. Comput. Sci. **385**(1-3) (2007) 34–48
13. Nichitiu, C., Rémila, É.: Leader election by d dimensional cellular automata. In Wiedermann, J., van Emde Boas, P., Nielsen, M., eds.: ICALP. Volume 1644 of Lecture Notes in Computer Science., Springer (1999) 565–574
14. Nichitiu, C.: Algorithmique sur graphes d'automates: élection d'un chef, simulations. PhD thesis, École Normale Supérieure de Lyon (1999)
15. Nichitiu, C., Mazoyer, J., Rémila, E.: Algorithms for leader election by cellular automata. Journal of Algorithms **41**(2) (2001) 302 – 329
16. Bacquey, N.: Leader election on two-dimensional periodic cellular automata. Theor. Comput. Sci. **659** (2017) 36–52
17. Angluin, D.: Local and global properties in networks of processors (extended abstract). In Miller, R.E., Ginsburg, S., Burkhard, W.A., Lipton, R.J., eds.: Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA, ACM (1980) 82–93
18. Chalopin, J.: Algorithmique Distribuée, Calculs Locaux et Homomorphismes de Graphes. PhD thesis, Université Bordeaux I (2006)
19. Boldi, P., Vigna, S.: Fibrations of graphs. Discrete Mathematics **243**(1-3) (2002) 21–66
20. Bender, M.A., Fernández, A., Ron, D., Sahai, A., Vadhan, S.: The power of a pebble: exploring and mapping directed graphs. Information and Computation **176**(1) (2002) 1–21 Extended abstract in Proceedings of 30th Annual ACM Symposium on Theory of Computing, pp. 269-278, Dallas, TX, May 1998.

21. Bacquey, N.: Complexity classes on spatially periodic cellular automata. In Mayr, E.W., Portier, N., eds.: STACS. Volume 25 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014) 112–124
22. Richard, G.: On the synchronisation problem over cellular automata. In Vollmer, H., Vallée, B., eds.: 34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany. Volume 66 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017) 54:1–54:13

A Algorithm for agents

Start
write init_pos
stay in place
Span Tree
repeat
  if MIRROR.vertex is not inside a tree **then**
    write CURRENT.port to children
    move to MIRROR.vertex
    write foster father to father
    move to SUCCESSOR of (CURRENT.vertex, foster father)
  else
    remove all colors except father,children, and init_pos
    move to SUCCESSOR
  end if
until CURRENT is intial
Look and Wait
write CURRENT wait marker .
repeat
  write step marker
  repeat
    move to SUCCESSOR
  until CURRENT has step marker
  remove step marker
  move to SUCCESSOR
until CURRENT has wait marker
remove wait marker
stay in place
Merge
repeat
  if —foster fathers— ¿ 1 **then**
    move to min($foster fathers$)
    while CURRENT.vertex is not a root **do**
      replace father with foster father
      move to old father
    end while
  end if
  write foster fathers to father
  move in father
  while father exists **do**
    move to father
  end while
until CURRENT.vertex is root
stay in place
Goto Span Tree