



HAL
open science

Self-Stabilizing Leader Election in Polynomial Steps

Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, Franck
Petit

► **To cite this version:**

Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, Franck Petit. Self-Stabilizing Leader Election in Polynomial Steps. 2014. hal-00980798v1

HAL Id: hal-00980798

<https://hal.science/hal-00980798v1>

Submitted on 18 Apr 2014 (v1), last revised 12 Feb 2015 (v4)

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

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

Self-Stabilizing Leader Election in Polynomial Steps*

Karine Altisen,[§] Alain Cournier,[†] Stéphane Devismes,[§] Anaïs Durand,[§] and Franck Petit[‡]

[§] Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France CNRS, VERIMAG, F-38000 Grenoble, France

[†] Univ. Picardie Jules Verne, MIS, Amiens, France

[‡] LIP6 UMR 7606, UPMC Sorbonne Universités, Paris

Abstract

In this paper, we propose a silent self-stabilizing leader election algorithm for bidirectional connected identified networks of arbitrary topology. This algorithm is written in the locally shared memory model. It assumes the distributed unfair daemon, the most general scheduling hypothesis of the model. Our algorithm requires no global knowledge on the network (such as an upper bound on the diameter or the number of processes, for example).

We show that its stabilization time is in $\Theta(n^3)$ steps in the worst case, where n is the number of processes. To the best of our knowledge, this is the first self-stabilizing leader election algorithm for arbitrary identified networks that is proved to achieve a stabilization time polynomial in steps.

Its memory requirement is asymptotically optimal, *i.e.*, $\Theta(\log n)$ bits per processes. Its round complexity is of the same order of magnitude — *i.e.*, $\Theta(n)$ rounds — as the best existing algorithms designed with similar settings (*i.e.*, those that do not use global knowledge and that are proved under the unfair daemon).

Keywords: Distributed algorithms, fault-tolerance, self-stabilization, leader election, unfair daemon.

1 Introduction

In distributed computing, the *leader election* problem consists in distinguishing one process, so-called the leader, among the others. We consider here identified networks. So, as it is usually done, we augment the problem by requiring all processes to eventually know the identifier of the leader. The leader election is fundamental as it is a basic component to solve many other important problems, *e.g.*, consensus, spanning tree constructions, implementing broadcasting and convergecasting methods, *etc.*

Self-stabilization [9, 10] is a versatile technique to withstand *any* transient fault in a distributed system: a self-stabilizing algorithm is able to recover, *i.e.*, reach a legitimate configuration, in finite time, regardless the *arbitrary* initial configuration of the system, and therefore also after the occurrence of transient faults. Thus, self-stabilization makes no hypotheses on the nature or extent of transient faults that could hit the system, and recovers from the effects of those faults in a unified manner. Such versatility comes at a price. After transient faults, there is a finite period of time, called the *stabilization phase*, before the system returns to a legitimate configuration. The *stabilization time* is then the worst case duration of the stabilization phase, *i.e.*, the maximum time to reach a legitimate configuration starting from an arbitrary one. Notice that efficiency of self-stabilizing algorithms is mainly evaluated according to their stabilization time and memory requirement.

In this paper, we consider the asynchronous silent self-stabilizing leader election problem in bidirectional, connected, and identified networks of arbitrary topology. We investigate solutions to this problem which are written in the locally shared memory model introduced by Dijkstra [9]. In this model, the (distributed) unfair daemon is known as the weakest scheduling assumption. Now, proving the self-stabilization of a given algorithm under such an assumption implies that the stabilization time is finite in terms of (atomic) steps. However, despite some solutions assuming all these settings (in particular

*This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

the unfairness assumption) are available in the literature [6, 7, 8], none of them is proven to achieve a polynomial upper bound in steps on the stabilization time. Rather, the time complexities of all these solutions are analyzed in terms of rounds only.

Related Work. In [11], Dolev *et al* showed that the silent self-stabilizing leader election requires $\Omega(\log n)$ bits per process, where n is the number of processes. Self-stabilizing leader election algorithms for arbitrary connected identified networks have been proposed in the message-passing model [1, 3, 4]. First, the algorithm of Afek and Bremner [1] stabilizes in $O(n)$ rounds using $\Theta(\log n)$ bits per process. But it assumes that the link-capacity is bounded by a value B , known by all processes. Two solutions that stabilize in $O(\mathcal{D})$ rounds, where \mathcal{D} is the diameter of the network, have been proposed in [3, 4]. However, both solutions assume that processes know some upper bound D on the diameter \mathcal{D} ; and have a memory requirement in $\Theta(\log D \log n)$ bits.

Several solutions are also given in the shared memory model [12, 2, 6, 7, 8, 13]. The algorithm proposed by Dolev and Herman [12] is not silent, works under a *fair* daemon, and assume that all processes know a bound N on the number of processes. This solution stabilizes in $O(\mathcal{D})$ rounds using $\Theta(N \log N)$ bits per process. The algorithm of Arora and Gouda [2] works under a *weakly fair* daemon and assume the knowledge of some bound N on the number of processes. This solution stabilizes in $O(N)$ rounds using $\Theta(\log N)$ bits per process.

Datta *et al* [6] propose the first self-stabilizing leader election algorithm (for arbitrary connected identified networks) proven under the (distributed) unfair daemon. This algorithm stabilizes in $O(n)$ rounds. However, the space complexity of this algorithm is unbounded. (More precisely, the algorithm requires each process to maintain an unbounded integer in its local memory.)

Solutions in [7, 8, 13] have a memory requirement which is asymptotically optimal (*i.e.* in $\Theta(\log n)$). The algorithm proposed by Kravchik and Kutten [13] assumes a synchronous daemon and the stabilization time of this latter is in $O(\mathcal{D})$ rounds. The two solutions proposed by Datta *et al* in [7, 8] assume a distributed unfair daemon and have a stabilization time in $O(n)$ rounds. However, despite these two algorithms stabilize within a finite number of step (indeed, they are proved assuming an unfair daemon), no step complexity analysis is proposed. Finally, note that the algorithm proposed in [7] assumes that each process has a bit of memory which cannot be arbitrarily corrupted.

Contribution. We propose a silent self-stabilizing leader election algorithm for arbitrary connected and identified networks. Our solution is written in the locally shared memory model assuming a distributed unfair daemon, the weakest scheduling assumption. Our algorithm assumes no knowledge of any global parameter (*e.g.*, an upper bound on \mathcal{D} or n) of network. Like previous solutions of the literature [7, 8], it is asymptotically optimal in space (*i.e.*, it works using $\Theta(\log n)$ bits per process), and it stabilizes in $\Theta(n)$ rounds in the worst case. Yet, contrary to those solutions, we show that our algorithm has a stabilization time in $\Theta(n^3)$ steps in the worst case.

For fair comparison, we have also studied the step complexity of the algorithm given in [8], this latter being the closest to ours in terms of performance. We show that the stabilization time of this algorithm is not polynomial in steps. In more details, fixing n to any arbitrary value, for every $\alpha \geq 3$, there exists a network of n processes in which there exists a possible execution that stabilizes in $\Omega(n^{\alpha+1})$ steps.

Roadmap. The next section is dedicated to the computational model and basic definitions. In Section 3, we propose our self-stabilizing leader election algorithm. We prove its correctness in Section 4. In the same section, we also study its stabilization time in both steps and rounds. We show that the stabilization time of the self-stabilizing leader election algorithm given in [8] is not polynomial in steps in Section 5. We conclude in Section 6.

2 Computational model

2.1 Distributed systems

We consider *distributed systems* made of n processes. Each process can communicate with a subset of other processes, called its *neighbors*. We denote by \mathcal{N}_p the set of neighbors of process p . Communications are assumed to be bidirectional, *i.e.* $q \in \mathcal{N}_p$ if and only if $p \in \mathcal{N}_q$. Hence, the topology of the system can

be represented as a simple undirected connected graph $G = (V, E)$, where V is the set of processes and E is a set of edges representing (direct) communication relations. We assume that each process has a unique ID, a natural integer. IDs are stored using a constant number of bits, say b . As commonly done in the literature, we assume that $b = \Theta(\log n)$. Moreover, by an abuse of notation, we identify a process with its ID, whenever convenient. We will also denote by ℓ the process of minimum ID.

2.2 Locally shared memory model

We consider the *locally shared memory model*, in which the processes communicate using a finite number of locally shared registers, called *variables*. Each process can read its own variables and those of its neighbors, but can write only to its own variables. The *state* of a process is the vector of the values of all its variables. A configuration γ of the system is the vector of the states of all processes. We denote by $\gamma(p)$ the state of the process p in the configuration γ . We denote by \mathcal{C} the set of all possible configurations.

A distributed *algorithm* consists of one *program* per process. The program of a process p is a finite set of actions of the following form:

$$\langle \text{label} \rangle :: \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

The *labels* are used to identify the actions. The *guard* of an action in the program of a process p is a Boolean expression involving the variables of p and its neighbors. If the guard of some action evaluates to true, then the action is said to be *enabled* at p . If at least one action is enabled at p , p is said to be enabled. We denote by $Enabled(\gamma)$ the set of processes enabled in configuration γ . The *statement* of an action is a sequence of assignments on the variables of p . An action can be executed only when it is enabled. In this case, the execution of the action consists in executing its statement.

The asynchronism of the system is materialized by an adversary, called the *daemon*. In a configuration γ , if $Enabled(\gamma) \neq \emptyset$, then the daemon selects a non empty subset S of $Enabled(\gamma)$ to perform an *atomic step*: $\forall p \in S$, p atomically executes one of its actions enabled in γ , leading the system to a new configuration γ' . We denote by \mapsto the relation between configurations such that $\gamma \mapsto \gamma'$ if and only if γ' can be reached from γ in one atomic step. An *execution* is then a *maximal* sequence of configurations $\gamma_0, \gamma_1, \dots$ such that $\gamma_{i-1} \mapsto \gamma_i, \forall i > 0$. The term “maximal” means that the execution is either infinite, or ends at a *terminal* configuration γ in which $Enabled(\gamma)$ is empty.

As we saw previously, each step from a configuration to another is driven by a daemon. In this paper, the daemon is supposed to be *distributed* and *unfair*. “Distributed” means that while the configuration is not terminal, the daemon should select at least one enabled process, maybe more. “Unfair” means that there is no fairness constraint, *i.e.*, the daemon might never permit an enabled process to execute, unless it is the only enabled process.

2.3 Rounds

To measure the time complexity of an algorithm, we also use the notion of *round*. This latter allows to highlight the time of execution according to the speed of the slowest process. If a process is enabled in a configuration γ_i but not enabled in the next configuration γ_{i+1} and does not execute any action between γ_i and γ_{i+1} , we said that p is *neutralized* during the step $\gamma_i \mapsto \gamma_{i+1}$. Let e be an execution. The minimal prefix of e such that every enabled process in γ_0 either executes at least one action or is neutralized is the first round of e , noted e' . Let e'' be the suffix of e starting from the last configuration of e' . The first round of e'' is the second round of e , and so on.

2.4 Self-Stabilization

Let \mathcal{A} be a distributed algorithm. Let \mathcal{E} be the set of all possible executions of \mathcal{A} . A *specification* SP is a predicate over \mathcal{E} .

\mathcal{A} is *self-stabilizing* for SP if and only if there exists a non-empty subset of configurations $\mathcal{L} \subseteq \mathcal{C}$, called *legitimate* configurations, such that:

- *Closure*: $\forall e \in \mathcal{E}$, for each step $\gamma_i \mapsto \gamma_{i+1} \in e$, $\gamma_i \in \mathcal{L} \Rightarrow \gamma_{i+1} \in \mathcal{L}$.
- *Convergence*: $\forall e \in \mathcal{E}, \exists \gamma \in e$ such that $\gamma \in \mathcal{L}$.

- *Correction*: $\forall e \in \mathcal{E}$ such that e starts in a legitimate configuration $\gamma \in \mathcal{L}$, e satisfies SP .

The *stabilization time* is the maximum time to reach a legitimate configuration starting from any configuration.

2.5 Self-Stabilizing Leader Election

We define $SP_{LE}(e)$ the specification of the leader election problem. Let $Leader : V \mapsto \mathbb{N}$ be a function defined on the state of any process $p \in V$ in the current configuration that returns the ID of the leader appointed by p . $SP_{LE}(e)$ is true if and only if:

1. For all configuration $\gamma \in e$, $\forall p, q \in V$, $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process in V .
2. For all step $\gamma_i \mapsto \gamma_{i+1} \in e$, $\forall p \in V$, $Leader(p)$ has the same value in γ_i and γ_{i+1} .

\mathcal{A} is *silent* if and only if every execution is finite [11]. Let γ be a terminal configuration. The set of all possible executions starting from γ is the singleton $\{\gamma\}$. So, if \mathcal{A} is self-stabilizing and silent, γ must be legitimate. Thus, to prove that a leader election algorithm is both self-stabilizing and silent, it is necessary and sufficient to show that:

- In every terminal configuration γ , $\forall p, q \in V$, $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process.
- Every execution is finite.

3 Algorithm \mathcal{LE}

In this section, we present a silent and self-stabilizing leader election algorithm, called \mathcal{LE} . Its formal code is given in Algorithm 1. Starting from an arbitrary configuration, \mathcal{LE} converges to a terminal configuration, where the process of minimum ID, ℓ , is elected. More precisely, in the terminal configuration, every process p knows the identifier of ℓ thanks to its local variable $p.idR$; moreover a spanning tree rooted at ℓ is defined using two variables per process: par and $level$. First, $\ell.par = \ell$ and $\ell.level = 0$. Then, for every process $p \neq \ell$, $p.par$ points to the parent of p in the tree and $p.level$ is the height of p in the tree.

First, we present a simple algorithm for the leader election problem in Subsection 3.1. We show why this algorithm is not self-stabilizing in Subsection 3.2. Then, we explain in Subsection 3.3 how to modify this simple algorithm to make it self-stabilizing.

3.1 Non Self-Stabilizing Leader Election

We first consider a simplified version of \mathcal{LE} . Starting from a predefined initial configuration, \mathcal{LE} elects ℓ in all idR variables and builds a spanning tree rooted at ℓ .

Initially, every process p declares itself as leader: $p.idR = p$, $p.par = p$, and $p.level = 0$. So, p satisfies the two following predicates:

$$SelfRoot(p) \equiv (p.par = p) \text{ and } SelfRootOk'(p) \equiv (p.level = 0) \wedge (p.idR = p)$$

Note that, in the sequel, we say that p is a *self root* when $SelfRoot(p)$ holds.

From such an initial configuration, our non self-stabilizing algorithm consists in the following single action:

$$\begin{aligned} J\text{-Action}' \quad &:: \quad \exists q \in \mathcal{N}_p, (q.idR < p.idR) \rightarrow \begin{aligned} &p.par = \min_{\leq} \{q \in \mathcal{N}_p\}; \\ &p.idR = p.par.idR; \\ &p.level = p.par.level + 1; \end{aligned} \end{aligned}$$

Informally, when p discovers that $p.idR$ is not equal to the minimum identifier, it updates its variables accordingly: let q be the neighbor of p having idR minimal. Then, p selects q as new parent ($p.par = q$ and $p.level = p.par.level + 1$) and sets $p.idR$ to the value of $q.idR$. If there are several neighbors having

Algorithm 1 Algorithm \mathcal{LE} for process p

Variables

$p.idR \in \mathbb{N}$
 $p.par \in \mathcal{N}_p \cup \{p\}$
 $p.level \in \mathbb{N}$
 $p.status \in \{C, EB, EF\}$

Macros

$Children_p \equiv \{q \in \mathcal{N}_p \mid q.par = p\}$
 $RealChildren_p \equiv \{q \in Children_p \mid KinshipOk(q, p)\}$
 $p \leq q \equiv (p.idR \leq q.idR) \wedge [(p.idR = q.idR) \Rightarrow (p \leq q)]$
 $Min_p \equiv \min_{\leq} \{q \in \mathcal{N}_p \mid q.status = C\}$

Predicates

$SelfRoot(p) \equiv p.par = p$
 $SelfRootOk(p) \equiv (p.level = 0) \wedge (p.idR = p) \wedge (p.status = C)$
 $GoodIdR(s, f) \equiv (s.idR \geq f.idR) \wedge (s.idR < s)$
 $GoodLevel(s, f) \equiv (s.idR = f.idR) \Rightarrow (s.level = f.level + 1)$
 $GoodStatus(s, f) \equiv [(s.status = EB) \Rightarrow (f.status = EB)]$
 $\quad \vee [(s.status = EF) \Rightarrow (f.status \neq C)]$
 $\quad \vee [(s.status = C) \Rightarrow (f.status \neq EF)]$
 $KinshipOk(s, f) \equiv GoodIdR(s, f) \wedge GoodLevel(s, f) \wedge GoodStatus(s, f)$
 $AbRoot(p) \equiv [SelfRoot(p) \wedge \neg SelfRootOk(p)]$
 $\quad \vee [\neg SelfRoot(p) \wedge \neg KinshipOk(p, p.par)]$
 $Allowed(p) \equiv \forall q \in Children_p, (\neg KinshipOk(q, p) \Rightarrow q.status \neq C)$

Guards

$EBroadcast(p) \equiv (p.status = C) \wedge [AbRoot(p) \vee (p.par.status = EB)]$
 $EFFeedback(p) \equiv (p.status = EB) \wedge (\forall q \in RealChildren_p, q.status = EF)$
 $Reset(p) \equiv (p.status = EF) \wedge AbRoot(p) \wedge Allowed(p)$
 $Join(p) \equiv (p.status = C) \wedge [\exists q \in \mathcal{N}_p, (q.idR < p.idR) \wedge (q.status = C)] \wedge Allowed(p)$

Actions

$EB\text{-action} \quad :: \quad EBroadcast(p) \quad \rightarrow \quad p.status = EB;$
 $EF\text{-action} \quad :: \quad EFFeedback(p) \quad \rightarrow \quad p.status = EF;$
 $R\text{-action} \quad :: \quad Reset(p) \quad \rightarrow \quad p.status = C;$
 $\quad \quad \quad \quad \quad \quad \quad \quad \quad p.par = p;$
 $\quad \quad \quad \quad \quad \quad \quad \quad \quad p.idR = p;$
 $\quad \quad \quad \quad \quad \quad \quad \quad \quad p.level = 0;$
 $J\text{-action} \quad :: \quad Join(p) \wedge \neg EBroadcast(p) \quad \rightarrow \quad p.par = Min_p;$
 $\quad \quad \quad \quad \quad \quad \quad \quad \quad p.idR = p.par.idR;$
 $\quad \quad \quad \quad \quad \quad \quad \quad \quad p.level = p.par.level + 1;$

idR minimal, we break ties using the identifiers of those neighbors. Thus, p selects q according to the lexical order \preceq defined as follows:

$$\forall p, q \in V, p \preceq q \Leftrightarrow (p.idR \leq q.idR) \wedge [(p.idR = q.idR) \Rightarrow (p \leq q)]$$

Hence, the identifier of ℓ is propagated, from neighbors to neighbors, into the idR variables and the system reaches a terminal configuration in $O(\mathcal{D})$ rounds. Figure 1 shows an example of such an execution.

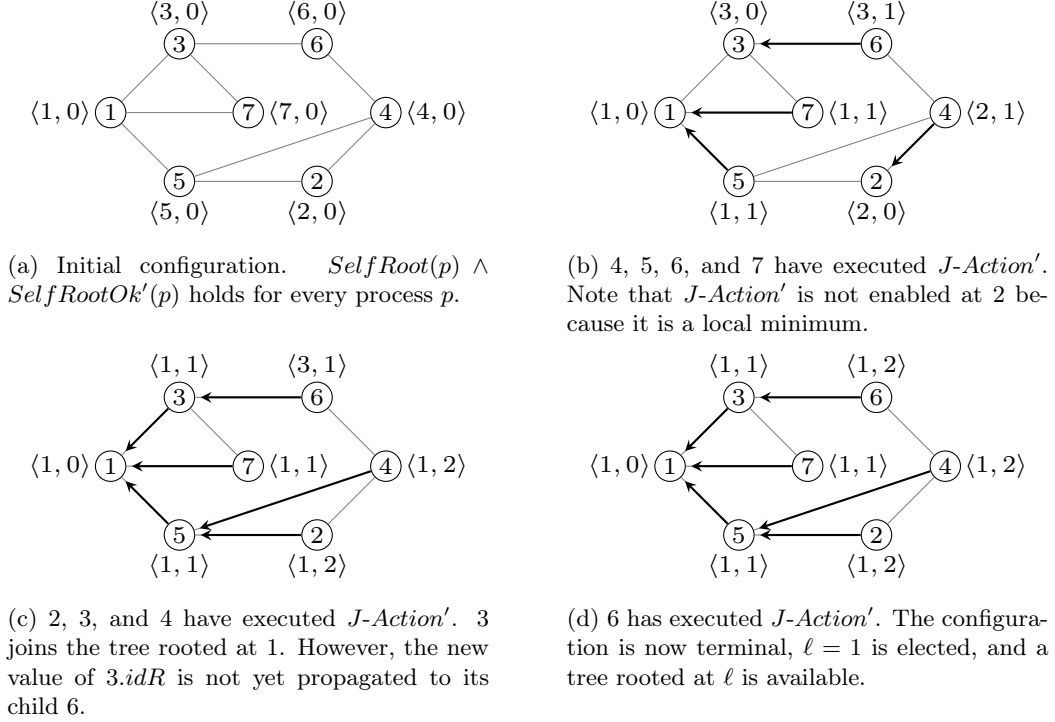


Figure 1: Example of execution of the non self-stabilizing algorithm. Process identifiers are given inside the nodes. $\langle x, y \rangle$ means $idR = x$ and $level = y$. Arrows represent par pointers. The absence of arrow means that the process is a self root.

Notice first that for every process p , $p.idR$ is always less or equal to its own identifier. Indeed, $p.idR$ is initialized to p and decreases each time p executes $J-Action'$. Hence, $p.idR = p$ while p is a self root and after p executes $J-Action'$ for the first time, $p.idR$ is strictly smaller than its ID forever.

Second, even in this simplified context, for each two neighbors p and q such that q is the parent of p , it may happens that $p.idR$ is greater than $q.idR$ —an example is shown in Figure 1c, where $p = 6$ and $q = 3$. This is due to the fact that p joins the tree of q but meanwhile q joins another tree and this change is not yet propagated to p . Similarly, when $p.idR \neq q.idR$, $p.level$ may be different from $q.level + 1$. According to those remarks, we can deduce that when $p.par = q$ with $q \neq p$, we have the following relation between p and q :

$$\begin{aligned} GoodIdR(p, q) &\equiv (p.idR \geq q.idR) \wedge (p.idR < p) \\ GoodLevel(p, q) &\equiv (p.idR = q.idR) \Rightarrow (p.level = q.level + 1) \end{aligned}$$

3.2 Fake IDs

This previous algorithm is not self-stabilizing. Indeed, in a self-stabilization context, the execution may start in an arbitrary configuration. In particular, idR variables can be initialized to arbitrary natural integer values, even values that are actually not IDs of (existing) processes. We call such values *fake IDs*.

The existence of fake IDs may lead the system to an illegitimate terminal configuration. Refer to the example of execution given in Figure 2: starting from Configuration 2a, if processes 3 and 4 move, the

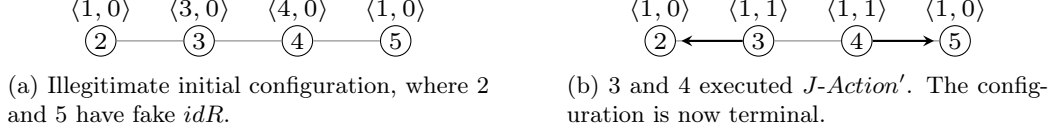


Figure 2: Example of an execution that does not converge to a legitimate configuration.

system reaches the terminal configuration given in 2b, where there are two trees and the idR variables elect the fake ID 1.

In this example, 2 and 5 can detect the problem. Indeed, predicate $SelfRootOk'$ is violated for both 2 and 5. One may believe that it is sufficient to reset the local state of 2 and 5 using the following action:

$$R\text{-Action}' \quad :: \quad SelfRoot(p) \wedge \neg SelfRootOk'(p) \rightarrow p.par = p; p.idR = p; p.level = 0;$$

Unfortunately, this may lead to an execution that never converges, as shown in Figure 3. Indeed, if a process resets, it becomes a self root but this does not erase the fake ID in the rest of its subtree. Then, another process can join the tree and adopt the fake ID which will further propagated, and so on. In the example, a process resets while another joins its tree at lower level, and this leads to endless erroneous behavior, since we do not want to assume any maximal value for $level$ (such an assumption would otherwise imply the knowledge of some upper bound on n). Therefore, the whole tree must be reset, instead of its root only. To that goal, we first froze the “abnormal” tree in order to forbid any process to join it, then the tree is reset top-down. The cleaning mechanism is detailed in the next subsection.

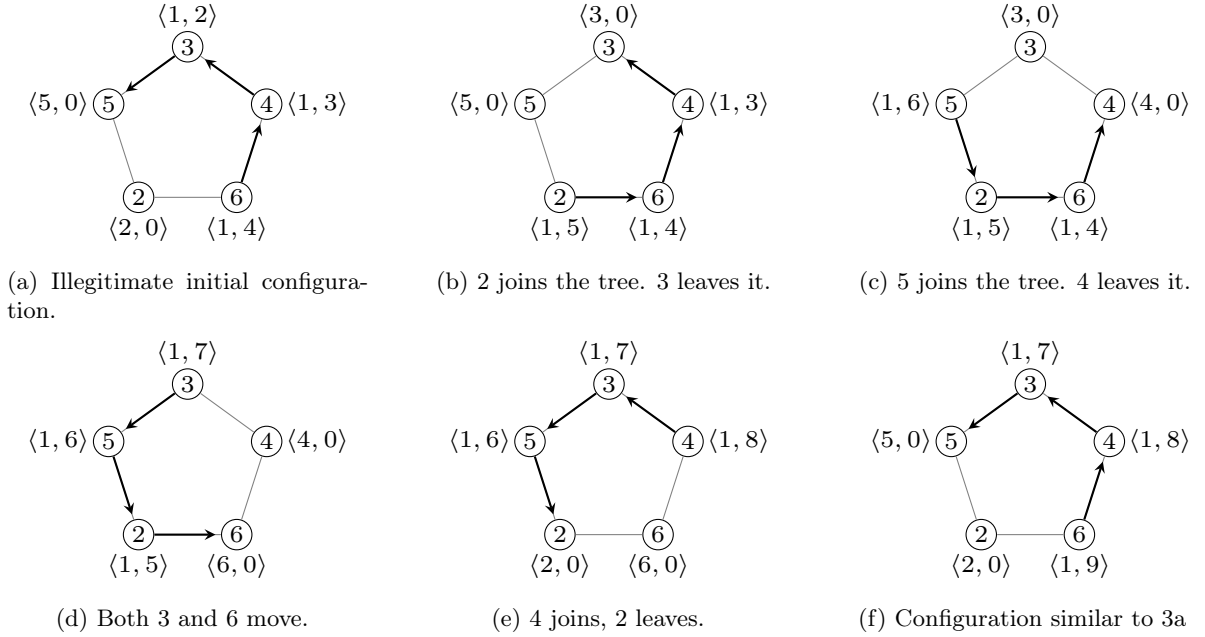


Figure 3: The first process of the chain violates the predicate $SelfRootOk'$ and resets by executing $R\text{-Action}'$, while another process joins its tree. This cycle of resets and joins might never terminate.

3.3 Cleaning Abnormal Trees

To detect possible errors (on idR , par , or $level$), we define what is a “good relation” between a parent and its children. Namely, the predicate $KinshipOk'(p, q)$ models that a process p is a *real child* of its parent $q = p.par$. This predicate holds if and only if $GoodLevel(p, q)$ and $GoodIdR(p, q)$ are true. This relation defines a spanning forest: a *tree* is a maximal set of processes connected by par pointers and

satisfying $KinshipOk'$ relation. A process p is a root of such a tree whenever $SelfRoot(p)$ holds or $KinshipOk'(p, p.par)$ is false. When $SelfRoot(p) \wedge SelfRootOk'(p)$ is true, p is a normal root just as in the non self-stabilizing case (see 3.1). In either case, there is an error and p is said to be an *abnormal root*:

$$AbRoot'(p) \equiv (SelfRoot(p) \wedge \neg SelfRootOk'(p)) \vee (\neg SelfRoot(p) \wedge \neg KinshipOk'(p, p.par))$$

A tree is called an *abnormal tree* when its root is abnormal.

We now detail the different predicates and actions of Algorithm 1.

Variable $status$. Abnormal trees need to be frozen before to be cleaned in order to prevent them from growing endlessly (see 3.2). This mechanism is achieved using an additional variable, $status$, that is used as follows. If a process p is clean (*i.e.*, not involved into any freezing operation), then its $p.status$ is C . Otherwise, it has status EB or EF and no neighbor can select it as its parent. These two latter states are actually used to perform a “Propagation of Information with Feedback” [5, 14] into the abnormal trees. Therefore, status EB means “Error Broadcast” and EF means “Error Feedback”. From an abnormal root, the status EB is broadcast down in the tree. Then, once the EB wave reaches a leaf, the leaf initiates a convergecast EF -wave. Once the EF -wave reaches the abnormal root, the tree is considered to be *dead*, meaning that there is no process of status C in the tree and no other process can join it. So, the tree can be safely reset from the abnormal root toward the leaves.

Notice that the new variable $status$ may also get arbitrary initialization. Thus, we enforce previously introduced predicates as follows.

A self root must have status C , otherwise it is an abnormal root:

$$SelfRootOk(p) \equiv SelfRootOk'(p) \wedge (p.status = C)$$

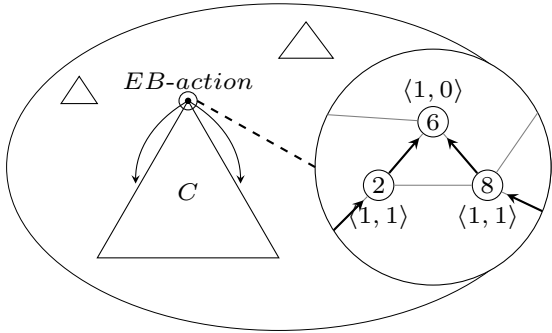
To be a real child of q , p should have a status coherent with one of q . This is expressed with the predicate $GoodStatus(p, q)$, which is used to enforce the $KinshipOk(p, q)$ relation:

$$\begin{aligned} GoodStatus(p, q) &\equiv [(p.status = EB) \Rightarrow (q.status = EB)] \vee [(p.status = EF) \Rightarrow (q.status \neq C)] \vee \\ &\quad [(p.status = C) \Rightarrow (q.status \neq EF)] \\ KinshipOk(p, q) &\equiv KinshipOk'(p, q) \wedge GoodStatus(p, q) \end{aligned}$$

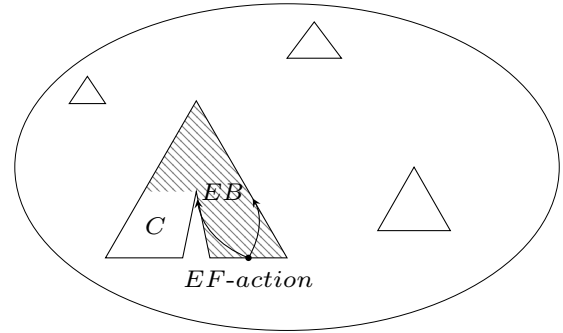
Precisely, when p has status C , its parent must have status C or EB (if the EB -wave is not propagated yet to p). If p has status EB , its parent must be of status EB because p gets status EB from its parent and its parent will change its status to EF only after p gets status EF . Finally, if p has status EF , its parent can have status EB (if the EF -wave is not propagated yet to its parent) or EF .

Normal Execution. Remark that, after all abnormal trees have been removed, all processes have status C and the algorithm works as in the initial version. Notice that the guard of J -action has been enforced so that only processes with status C and which are not abnormal root can execute it, and when executing J -action, a process can only choose a neighbor of status C as parent. Moreover, remark that the cleaning of all abnormal trees does not ensure that all fake IDs have been removed. Rather, it guarantees the removal of all fake IDs smaller than ℓ . This implies that (at least) ℓ is a self root at the end of the cleaning and all other processes will elect ℓ within the next \mathcal{D} rounds.

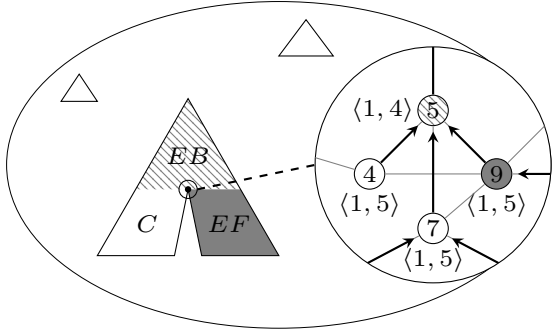
Cleaning Abnormal Trees. Figure 4 shows how an abnormal tree is cleaned. In the first phase (see Figure 4a), the root broadcasts status EB down to its (abnormal) tree: all the processes in this tree execute EB -action, switch to status EB and are consequently informed that they are in an abnormal tree. The second phase starts when the EB -wave reaches a leaf. Then, a convergecast wave of status EF is initiated thanks to action EF -action (see Figure 4b). The system is asynchronous, hence all the processes along some branch can have status EF before the broadcast of the EB -wave is done into another branch. In this case, the parent of these two branches waits that all its children in the tree (processes in the set $RealChildren$) get status EF before executing EF -action (Figure 4c). When the root gets status EF , all processes have status EF : the tree is dead. Then (third phase), the root can reset (safely) to become a self root by executing R -action (Figure 4e). Its former real children (of status EF) become themselves abnormal roots of dead trees (Figure 4f) and reset, *etc.*



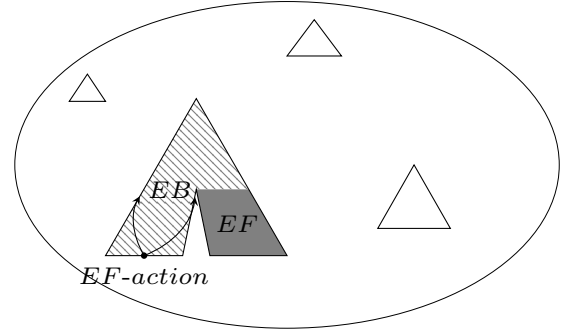
(a) When an abnormal root detects an error, it executes *EB-action*. The *EB-wave* is broadcast to the leaves. Here, 6 is an abnormal root because it is a self root and its *idR* is different from its ID ($1 \neq 6$).



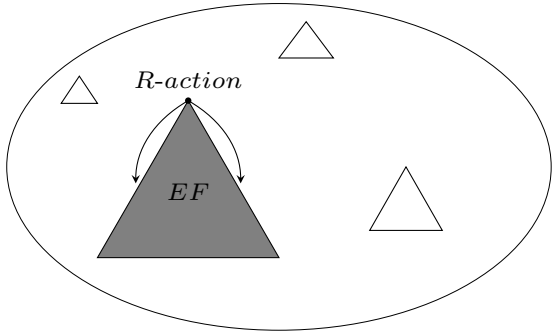
(b) When the *EB-wave* reaches a leaf, it executes *EF-action*. The *EF-wave* is propagated up to the root.



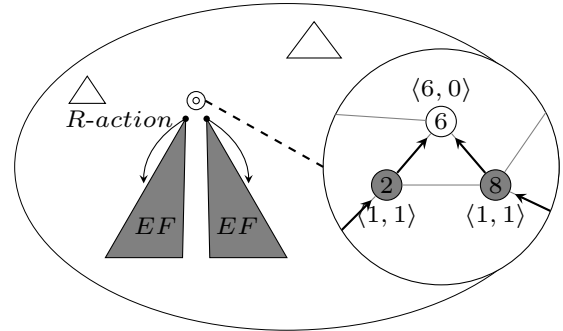
(c) It may happen that the *EF-wave* reaches a node, here process 5, even though the *EB-wave* is still broadcasting into some of its proper subtrees: 5 must wait that the status of 4 and 7 become *EF* before executing *EF-action*.



(d) *EB-wave* has been propagated in the other branch. An *EF-wave* is initiated by the leaves.

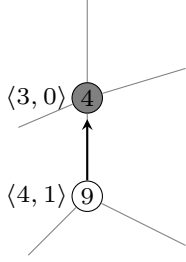


(e) *EF-wave* reaches the root. The root can safely reset (*R-action*) because its tree is dead. The cleaning wave is propagated down to the leaves.

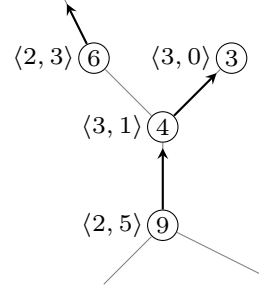


(f) Its children become themselves abnormal roots of dead trees and can execute *R-action*: 2 and 8 can clean because their status is *EF* and their parent has status *C*.

Figure 4: Schematic example of the cleaning mechanism. Trees are filled according to the status of their processes: white for *C*, dashed for *EB*, gray for *EF*.



(a) 4 and 9 are abnormal roots. If 4 executes *R-action* before 9 executes *EB-action*, the kinship relation between 4 and 9 becomes correct and 9 is no more an abnormal root. Then, *EB-action* is no more enabled at 9.



(b) 9 is an abnormal root and Min_4 is 6. If 4 executes *J-action* before 9 executes *EB-action*, the kinship relation between 4 and 9 becomes correct and 9 is no more an abnormal root. Then, *EB-action* is no more enabled at 9.

Figure 5: Example of situations where the parent of a process is locked.

Finally, we used the predicate $Allowed(p)$ to temporarily lock the parent of p in two particular situations, illustrated in Figure 5, when p is enabled to switch its status from *C* to *EB*. These locks impact neither the correctness nor the complexity of \mathcal{LE} . Rather, they allow us to simplify the proofs by ensuring that, once enabled, *EB-action* remains continuously enabled until executed.

4 Correctness and Complexity Analysis

In this section, we first define some concepts which will be used in the proofs (Subsection 4.1). Then, we show in Subsection 4.2 that Algorithm \mathcal{LE} is self-stabilizing and silent for the leader election, assuming a distributed unfair daemon. Along the proof, we also establish a bound on its stabilization time in steps, namely $O(n^3)$. Finally, we study more precisely the complexity of \mathcal{LE} in Subsection 4.3 (in particular, we give its complexity in rounds).

4.1 Some definitions

First, we instantiate the function $Leader(p)$ used in the specification of the leader election (Section 2.5).

Definition 1 (Leader). For each process p , for every configuration γ , the value $Leader(p)$ in γ is $p.idR$.

Note that the value of $Leader(p)$ depends on the current configuration γ . Nevertheless, when it is clear from the context, we omit the mention to γ . This will be also the case for every predicates and notations used in the sequel.

We now recall some definitions and notations from graph theory. A *path* \mathcal{P} , from p_k to p_0 is a sequence of processes p_k, p_{k-1}, \dots, p_0 such that $p_{i-1} \in \mathcal{N}_{p_i}$, for all i in $\{1, \dots, k\}$. Nodes p_k and p_0 are respectively called the *initial* and *terminal extremity* of \mathcal{P} . The length of \mathcal{P} , denoted by $|\mathcal{P}|$, is equal to k . We call *cycle* any path p_k, p_{k-1}, \dots, p_0 such that $p_0 = p_k$. The distance between two processes p and q , denoted $\|p, q\|$, is equal to the length of the shortest path between p and q . The *diameter* of the network, denoted \mathcal{D} , is the maximum distance between any two processes.

The rest of the paragraph is dedicated to introducing and justifying the notion trees induced by the *KinshipOk* relation. We first show that the predicate *KinshipOk* is an acyclic relation. To that goal, we define the graph induced by the *KinshipOk* relation.

Definition 2 (Kinship Relation Graph). For some configuration γ , let $G_{kr} = (V, KR)$ be a directed graph such that $(p, q) \in KR \Leftrightarrow (\{p, q\} \in E) \wedge (p.par = q) \wedge KinshipOk(p, q)$. G_{kr} is called the graph of *kinship relations* in γ .

We first show that G_{kr} is a DAG (Directed Acyclic Graph). We recall, *path* and *cycle* naturally extend to directed graph, *i.e.*, a (directed) path \mathcal{P} in G_{kr} is a sequence of processes p_k, p_{k-1}, \dots, p_0 such that $(p_{i+1}, p_i) \in KR$, for all i in $\{0, \dots, k-1\}$.

Lemma 1. *Let γ be a configuration. The graph of kinship relations in γ contains no cycle.*

Proof. By definition, for all pairs of processes p, q such that $KinshipOk(p, q)$ holds, we have: $p.idR \geq q.idR$ and $p.idR = q.idR \Rightarrow p.level = q.level + 1$. Hence, the processes along any path in G_{kr} are ordered w.r.t. the strict lexical order on the pair $(idR, level)$. The result directly follows. \square

Hence G_{kr} is a DAG (Directed Acyclic Graph) and even a spanning forest since the condition $p.par = q$ implies at most one successor per process in KR . Below, we define the roots and trees of this spanning forest.

Definition 3 (Root). For some configuration γ , a process p satisfies $Root(p)$ (and is called a *root* in γ) if and only if $SelfRoot(p) \vee AbRoot(p)$, or equivalently $SelfRoot(p) \vee \neg KinshipOk(p, p.par)$ holds in γ .

Next, we define the paths, called $KPaths$, that follow the tree structures in G_{kr} , *i.e.*, the paths linking each process to the root of its own tree.

Definition 4 (KPath). For every process p , $KPath(p)$ is the unique path p_0, p_1, \dots, p_k such that $p_k = p$ and satisfying the following conditions:

- $\forall i, 1 \leq i \leq k, (p_i.par = p_{i-1}) \wedge KinshipOk(p_i, p_{i-1})$
- $Root(p_0)$

Using Definitions 3 and 4, we formally define trees as follows.

Definition 5 (Tree). For some configuration γ , for every process p such that $Root(p)$, we define $Tree(p)$, the tree rooted at p , as follows:

$$Tree(p) = \{q \in V \mid p \text{ is the initial extremity of } KPath(q)\}$$

We give in Observation 1 an invariant on $KPaths$ when looking at the status of the processes. This property is based on the notion of S -Trace defined below.

Definition 6 (S -Trace). For some configuration γ , for a sequence of processes p_0, p_1, \dots, p_k , we define $S\text{-Trace}(p_0, p_1, \dots, p_k) \in \{C, EB, EF\}^*$ as the sequence $(p_0.status).(p_1.status) \dots (p_k.status)$ in γ .

Observation 1. *For any configuration, we have: $\forall p \in V, S\text{-Trace}(KPath(p)) \in EB^*C^* \cup EB^*EF^*$.*

Proof. Let p be a process. If $|KPath(p)| = 1$, Observation 1 trivially holds. For $|KPath(p)| \geq 2$, assume by contradiction that $S\text{-Trace}(KPath(p)) \notin EB^*C^* \cup EB^*EF^*$. Then $\exists s, f \in KPath(p)$ such that $s.par = f$ and $S\text{-Trace}(f, s) \in \{C.EB, C.EF, EF.EB, EF.C\}$. In all cases, $\neg GoodStatus(s, f)$ holds, which in turns implies that $\neg KinshipOk(s, f)$. This contradicts Definition 4. \square

4.2 Correctness

To prove the self-stabilization of Algorithm \mathcal{LE} under an unfair daemon, we first show that any execution is finite (Theorem 1) and then we show that in any terminal configuration, there is a unique leader: for every two processes, p and q , we have $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process (Theorem 2).

4.2.1 Termination of \mathcal{LE}

The goal, here, is to show that any execution contains a finite number of steps. We first partition a given execution into a finite number of segments (Lemma 4), see Fig. 6. Then, we prove that each segment contains a finite number of J -actions (Lemma 10). This latter result implies that every execution contains a finite number of J -actions (Corollary 2). Then, we show, in Corollary 3 and Lemma 11, that every execution contains a finite number of other actions. This allows us to conclude in Theorem 1 that every execution contains a finite number of steps.

Abnormal Trees. First, we introduce some notions that refine the concept of trees.

Definition 7 (Normal/Abnormal Tree). For some configuration γ and some process p , any tree rooted at p such that $\neg AbRoot(\gamma(p))$ is called a *normal tree* in γ . By Definition 3, $SelfRoot(\gamma(p)) \wedge SelfRootOk(\gamma(p))$ holds in this case. Any other tree is called an *abnormal tree*.

Definition 8 (Alive/Dead). Let γ be a configuration. A process p is called *alive* in γ if and only if $\gamma(p).status = C$. Otherwise it is said to be *dead*. A tree rooted at some process r , $Tree(r)$, in γ is called an *alive tree* in γ if and only if $\exists p \in Tree(r)$ such that p is alive in γ . Otherwise, it is called a *dead tree*.

Definition 9 (Leave/Join a Tree). Let $\gamma \mapsto \gamma'$ be a step. If a process p is in a tree T in γ and is not in the same tree T' in γ' ($T \neq T'$), we say that p *leaves* T and *joins* T' in $\gamma \mapsto \gamma'$.

Remark 1. No process can join a dead tree.

Lemma 2. *No alive abnormal root can be created.*

Proof. Let p be a process which is not an alive abnormal root in some configuration γ . This means that p is dead or p is a normal root ($SelfRoot(\gamma(p)) \wedge SelfRootOk(\gamma(p))$ holds) or p is not a root ($KinshipOk(\gamma(p), \gamma(p).par)$ holds).

Let $\gamma \mapsto \gamma'$ be a step. If p executes *EB-action* (respectively *EF-action*) during the step $\gamma \mapsto \gamma'$ then $\gamma'(p).status = EB$ (respectively $\gamma'(p).status = EF$) and p is dead in γ' .

If p executes *R-action*, $\gamma'(p).par = p$ and, consequently, $SelfRoot(\gamma'(p)) \wedge SelfRootOk(\gamma'(p))$ holds, consequently, p is a normal root in γ' .

If p executes *J-action*, let $q = Min_p$ in γ . By definition of *J-action*, $\gamma(q).status = C$, $\gamma(p).status = \gamma'(p).status = C$ and $\gamma(p).idR \leq p$ (since p is not an abnormal root at γ). Also, $\neg SelfRoot(\gamma'(p))$ holds.

- If q does not move in $\gamma \mapsto \gamma'$, then $\gamma'(p).par = q$, $\gamma'(q).status = C = \gamma'(p).status$, $\gamma'(p).level = \gamma(q).level + 1 = \gamma'(q).level + 1$, $\gamma'(p).idR = \gamma(q).idR = \gamma'(q).idR < \gamma(p).idR \leq p$. Hence, $KinshipOk(\gamma'(p), \gamma'(p).par)$ is true: p is not a root in γ' .
- Assume now that q moves in $\gamma \mapsto \gamma'$. As $\gamma(q).status = C$, q can only execute *EB-action* or *J-action* in the step. Consequently, $\gamma'(q).idR \leq \gamma(q).idR$.

Then, $\gamma'(p).par = q \neq p$, so $\neg SelfRoot(\gamma'(p))$ holds. $\gamma'(p).idR = \gamma(q).idR \geq \gamma'(q).idR$ and $\gamma'(p).idR = \gamma(q).idR < \gamma(p).idR \leq p$. So, $GoodIdR(\gamma'(p), \gamma'(q))$ holds.

If q executes *J-action*, $\gamma'(p).idR \neq \gamma'(q).idR$. Otherwise, $\gamma'(p).idR = \gamma'(q).idR$ and $\gamma'(p).level = \gamma(q).level + 1 = \gamma'(q).level + 1$. So $GoodLevel(\gamma'(p), \gamma'(q))$ holds.

Finally, $\gamma'(p).status = \gamma(p).status = C$ and $\gamma'(q).status \in \{C, EB\}$, so $GoodStatus(\gamma'(p), \gamma'(q))$ holds.

Thus, $KinshipOk(\gamma'(p), \gamma'(q))$ is true and p is not a root in γ' .

Assume now that p executes no action in the step $\gamma \mapsto \gamma'$. The only way for p to become an alive abnormal root is that $\gamma(p).par$ moves during the step, since the property “alive abnormal root” only depends on p and $p.par$. Furthermore, as p is not an alive abnormal root, when p is a normal root in γ , it stays so, in γ' .

Therefore, let us consider the case when p is not a root in γ and $\gamma(p).par$ moves. As p changes none of its variables, the only way for it to become an alive abnormal root is to have status C in γ and thus in γ' . As $GoodStatus(\gamma(p), \gamma(p).par)$ holds, this implies that the status of $\gamma(p).par$ is either EB or C . Looking at the EB case, p is a real child of $\gamma(p).par$ with status C ; hence *EF-action* is disabled for $\gamma(p).par$. Looking at the C case, $\gamma(p).par$ can execute *EB-action* and can change only its status to EB : $GoodStatus(\gamma'(p), \gamma'(p).par)$ holds and consequently $KinshipOk(\gamma'(p), \gamma'(p).par)$. $\gamma(p).par$ can also execute *J-action* in γ . This means that in γ and γ' , $p.par$ has status C , hence $GoodStatus(\gamma'(p), \gamma'(p).par)$ holds. Furthermore, $p.par$ takes a smaller value of idR in γ' , hence $GoodIdR(\gamma'(p), \gamma'(p).par)$ and $GoodLevel(\gamma'(p), \gamma'(p).par)$ are verified, and consequently $KinshipOk(\gamma'(p), \gamma'(p).par)$. \square

Lemma 3. *No alive abnormal tree can be created.*

Proof. Let $\gamma \mapsto \gamma'$ a step. Let $p \in V$. Assume there is no alive abnormal tree rooted at p in γ . In particular, p is not an alive abnormal root in γ . Then, assume, by contradiction, that $Tree(p)$ exists and is an alive abnormal tree in γ' .

- If $\gamma'(p).status = EF$, then every process in the tree has status EF (Observation 1) and the tree is dead, a contradiction.
- If $\gamma'(p).status = C$, then p is an alive abnormal root in γ' . But no alive abnormal root is created (Lemma 2), a contradiction.
- If $\gamma'(p).status = EB$. Then, according to the algorithm, there are two possible cases:

$\gamma(p).status = EB$:

- If $AbRoot(\gamma(p))$, then $Tree(p)$ is dead in γ (otherwise, $Tree(p)$ is an abnormal alive tree in γ , a contradiction). By the definition of J -action, no process can join $Tree(p)$ in $\gamma \mapsto \gamma'$. Moreover, as $\gamma(p).status = EB$, no process q in $Tree(p)$ satisfies $Reset(q)$ in γ , by Observation 1. Consequently, no process can leave $Tree(p)$ in $\gamma \mapsto \gamma'$. So, every process in $Tree(p)$ still have status EF or EB in γ' , i.e. $Tree(p)$ is still dead in γ' , a contradiction.
- If $\neg AbRoot(\gamma(p))$, then p does not satisfy $SelfRoot(p)$, otherwise $SelfRootOk(p)$ implies that $\gamma(p).status = C$, a contradiction. So, let $q = \gamma(p).par \in \mathcal{N}_p$. $\neg AbRoot(\gamma(p))$ implies that $q.status = EB$ and $KinshipOk(p, q)$ in γ . This latter also implies that $p \in RealChildren_q$ in γ . Now, $p \in RealChildren_q$ and $p.status = EB$ in γ implies that q is disabled in γ . Moreover, as $\gamma'(p).status = EB$, p does not execute any action in $\gamma \mapsto \gamma'$. So, $\neg AbRoot(p)$ still holds in γ' , a contradiction.

$\gamma(p).status = C$: Then, $\neg AbRoot(\gamma(p))$ (otherwise p is an abnormal alive root in γ). Then, p executes EB -action in $\gamma \mapsto \gamma'$ to get status EB . So, $EBroadcast(p) \wedge \neg AbRoot(p)$ implies that $p.par \neq p$ and $p.par.status = EB$ in γ . So, let $q = \gamma(p).par \in \mathcal{N}_p$. Now $p.par \neq p \wedge \neg AbRoot(p)$ implies that $KinshipOk(p, q)$ in γ . So, $p \in RealChildren_q$ and, as $p.status = C$ and $q.status = EB$ in γ , q is disabled in γ . Moreover, as $\gamma'(p).status = EB$, p necessarily executes EB -action in $\gamma \mapsto \gamma'$, which only changes its status to EB . So, $\neg AbRoot(p)$ still holds in γ' , a contradiction.

□

Finite Number of J -actions. To show that every process p executes only a finite number of J -actions, we prove below that p can only execute a finite number of J -actions in each segment of execution — a segment being separated from its follower by the death or the disappearance of some tree.

Definition 10 (Disappear/Die). Let $\gamma \mapsto \gamma'$ be some step and let p be a process such that $Root(p)$ in γ .

$Tree(p)$ disappears during the step $\gamma \mapsto \gamma'$ if and only if $Tree(p)$ is no more defined in γ' , namely $\neg Root(\gamma'(p))$.

$Tree(p)$ dies during the step $\gamma \mapsto \gamma'$ if and only if $Tree(p)$ is alive in γ and $Tree(p)$ exists, namely $Root(\gamma'(p))$, and is dead in γ' .

Definition 11 (Segment of execution). Let $e = \gamma_0\gamma_1\dots$ be any execution. $e' = \gamma_i\dots\gamma_j$ is a *segment of execution* e (segment, for short) if and only if e' is a maximal factor of e , where no tree dies nor disappears.

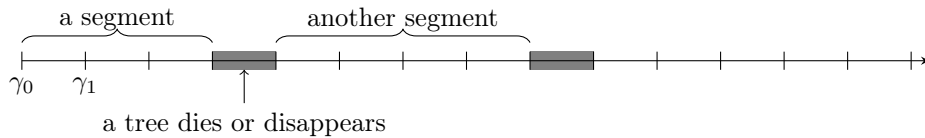


Figure 6: Segments of execution

Figure 6 illustrates Definition 11. We now show that the number of segments is finite.

Lemma 4. *There are at most $n + 1$ segments in any execution.*

Proof. In the initial configuration, there are at most n abnormal roots (every process) and, consequently, at most n abnormal trees. As no alive abnormal tree can be created (Lemma 3), if an abnormal tree is alive, then it is alive since the initial configuration. So, there is at most n trees that die or disappear and, consequently, there are at most $n + 1$ segments in the execution. \square

We now count the number of J -actions processes can execute in a given segment. For that purpose, we first need to prove intermediate lemmas that identify properties on computation steps.

Observation 2. *Let γ be a configuration and let p a process such that $Reset(p)$ is true in γ . Then $Tree(p)$ exists and is dead in γ .*

Proof. Let γ be a configuration and let p be a process such that $Reset(p)$ is true in γ . By definition, $AbRoot(\gamma(p))$ holds, hence $Tree(p)$ is defined in γ . Furthermore, $\gamma(p).status = EF$: by Observation 1, every process in $Tree(p)$ has status EF in γ , and we are done. \square

Lemma 5. *Let $\gamma \mapsto \gamma'$ be a step and let p be a process such that $p.status \in \{EB, EF\}$ in γ . We note r the root of the tree which contains p in γ . $Tree(r)$ is an abnormal tree in γ . And if $Tree(r)$ does not disappear during the step $\gamma \mapsto \gamma'$, p is still in $Tree(r)$ in γ' unless $Tree(r)$ was dead in γ .*

Proof. Let $\gamma \mapsto \gamma'$ be a step and let p be a process such that $p.status \in \{EB, EF\}$ in γ . We note r the root of the tree containing p in γ . As $S-Trace(KPath(p)) \in EB^*EF^*$, by Observation 1, the status of r in γ is either EF or EB . Hence $AbRoot(\gamma(r))$ holds: $Tree(r)$ is an abnormal tree in γ .

Assume now that $Root(\gamma'(r))$ holds (the tree does not disappear during the step). If r executes R -action in $\gamma \mapsto \gamma'$, Observation 2 applies in γ and proves that $Tree(r)$ is dead in γ .

If r does not (or cannot) execute its only possible action, R -action, as $Root(\gamma'(r))$ holds, r is still abnormal root in γ' . Let then $q \in KPath(\gamma(p))$ with $q \neq r$. By Observation 1, $\gamma(q).status \in \{EB, EF\}$ also. If $\gamma(q).status = EB$, q can only execute EF -action and if $\gamma(q).status = EF$, q is disabled, as $q \neq r$. Executing EF -action preserves $GoodStatus$ and hence $KinshipOk$ relations. Therefore, the $KPath$ from p to r is the same in γ and γ' and then $p \in Tree(r)$ in γ' . \square

Lemma 6. *Let p be a process and let $\gamma \mapsto \gamma'$ be step. If p is an abnormal root of status C in γ , then it is still an abnormal root in γ' .*

Proof. Let $\gamma \mapsto \gamma'$ be step and let p be a process such that $AbRoot(\gamma(p)) \wedge \gamma(p).status = C$: p can only execute EB -action. Therefore, $\gamma'(p).status \in \{C, EB\}$ and every other variable of p has identical value in γ and γ' .

So, if $SelfRoot(\gamma(p))$, then $\neg SelfRootOk(\gamma(p))$, and still $SelfRoot(\gamma'(p)) \wedge \neg SelfRootOk(\gamma'(p))$.

Otherwise, $\gamma(p).par \neq p$, $\neg SelfRoot(\gamma(p))$ and still $\neg SelfRoot(\gamma'(p))$. Let $q = \gamma(p).par$ and consider the following cases:

$\gamma(q).status = EF$: Then, $\neg GoodStatus(\gamma(p), \gamma(q))$, which implies $\neg KinshipOk(\gamma(p), \gamma(q))$. However, $p \in Children_{\gamma(q)}$. So, $\neg Allowed(q)$ holds in γ , and q is disabled. So, $\gamma'(p).status \in \{C, EB\}$ and $\gamma'(q).status = EF$, which implies $\neg GoodStatus(\gamma'(p), \gamma'(q))$. Thus, $\neg KinshipOk(\gamma'(p), \gamma'(q))$.

$\gamma(q).status = EB$: Then, $GoodStatus(\gamma(p), \gamma(q))$. So, $AbRoot(\gamma(p))$ implies that $\neg GoodIdR(\gamma(p), \gamma(q)) \vee \neg GoodLevel(\gamma(p), \gamma(q))$ holds. Now, q can only executes EF -action in $\gamma \mapsto \gamma'$. So, neither p nor q modify their variables par , idR , or $level$ in $\gamma \mapsto \gamma'$, and, consequently, $\neg GoodIdR(\gamma'(p), \gamma'(q)) \vee \neg GoodLevel(\gamma'(p), \gamma'(q))$ still holds. So, $\neg KinshipOk(\gamma'(p), \gamma'(q))$ in γ' .

$\gamma(q).status = C$: $AbRoot(\gamma(p))$ implies $\neg KinshipOk(\gamma(p), \gamma(q))$. Thus, $\neg Allowed(\gamma(q))$ holds because $p \in Children_{\gamma(q)}$ and $\gamma(p).status = C$. So, q cannot execute J -action in $\gamma \mapsto \gamma'$.

Then, as $\gamma(q).status = C \wedge \gamma(p).status = C$, we have $GoodStatus(\gamma(p), \gamma(q))$. So, $AbRoot(\gamma(p))$ implies that $\neg GoodIdR(\gamma(p), \gamma(q)) \vee \neg GoodLevel(\gamma(p), \gamma(q))$ holds. As p and q can only modify their status in $\gamma \mapsto \gamma'$ (q can only execute EB -action in $\gamma \mapsto \gamma'$), $\neg GoodIdR(\gamma'(p), \gamma'(q)) \vee \neg GoodLevel(\gamma'(p), \gamma'(q))$ still holds. So, $\neg KinshipOk(\gamma'(p), \gamma'(q))$.

In any cases, $\neg KinshipOk(\gamma'(p), \gamma'(q))$. As $\neg SelfRoot(\gamma'(p))$, $AbRoot(\gamma'(p))$ holds. \square

Lemma 7. *Let γ be a configuration and let p be a process such that $p.status \in \{EB, EF\}$ in γ . We note r the root of the tree which contains p in γ . Let γ_R be the first configuration, if any, after γ , such that p executes an R -action $\gamma_R \mapsto \gamma_{R+1}$.*

Assume γ_R exists, then $Tree(r)$ is dead in γ_R or has disappeared (at least once) between γ and γ_R .

Proof. Let γ be a configuration and let p be a process such that $p.status \in \{EB, EF\}$ in γ . We note r the root of the tree which contains p in γ . Let $\gamma = \gamma_0\gamma_1\dots$ be an execution starting at γ . Let γ_R be the first configuration, if any, in this execution such that p executes an R -action during the step $\gamma_R \mapsto \gamma_{R+1}$.

For every configuration γ_x , $x \in \{0, \dots, R-1\}$, the status of p is EB or EF . Hence, Lemma 5 applies iteratively in γ_x : either $Tree(r)$ disappears during the step $\gamma_x \mapsto \gamma_{x+1}$, or, if not, $p \in Tree(r)$ in γ_{x+1} . Hence, in γ_R , either $Tree(r)$ has disappeared or, if not, $p \in Tree(r)$.

When $p \in Tree(r)$ in γ_R , by assumption, p executes an R -action between γ_R and γ_{R+1} . Hence, $AbRoot(p)$ holds in γ_R and thus $p = r$. Furthermore, Observation 2 applies and proves that $Tree(r)$ is dead in γ_R . \square

Lemma 8. *Let p be a process and let $\gamma \mapsto \gamma'$ be a step. We note r the root of the tree which contains p in γ . If $EBroadcast(p)$ holds in γ , then $Tree(r)$ is an abnormal alive tree in γ and if $Tree(r)$ has not disappeared in γ' , p still belongs to $Tree(r)$ in γ' .*

Proof. Let $\gamma \mapsto \gamma'$ be a step. Let $p \in V$ such that $EBroadcast(\gamma(p))$.

If $AbRoot(\gamma(p))$ holds, then $p = r$ is the root of an alive abnormal tree, since $\gamma(p).status = C$. Furthermore, if $Tree(p)$ exists in γ' , $p \in Tree(p)$ in γ' , trivially.

Otherwise $\neg AbRoot(\gamma(p))$, $\gamma(p).par.status = EB$ and $KinshipOk(\gamma(p), \gamma(p).par)$. Applying Lemma 5 to $\gamma(p).par$, we have that $\gamma(p).par$ belongs to an abnormal alive tree in γ and so does p : $Tree(r)$ is an alive abnormal tree.

Furthermore, first note that $\gamma(p).par = \gamma'(p).par$. So, still by Lemma 5, in γ' , if $Tree(r)$ exists in γ' , $\gamma'(p).par$ belongs to $Tree(r)$ in γ' , since $Tree(r)$ is not dead in γ ($\gamma(p).status = C$). As $KinshipOk(\gamma(p), \gamma(p).par)$, we have that $p \in RealChildren_{\gamma(q)}$. Since $\gamma(p).status = C$, q is disabled in γ (because of p) and, as p only modifies its status to EB in $\gamma \mapsto \gamma'$, we still have $p \in RealChildren_{\gamma'(q)}$, i.e., p and q belong to the same abnormal tree, $Tree(r)$, in γ' . \square

Corollary 1. *Let γ be a configuration and let p be a process such that $EBroadcast(p)$ holds in γ . We note r the root of the tree which contains p in γ . Let γ_R be the first configuration, if any, since γ , such that p executes an R -action $\gamma_R \mapsto \gamma_{R+1}$.*

Assume γ_R exists. The tree $Tree(r)$ is an alive abnormal tree in γ but it is dead in γ_R or has disappeared (at least once) between γ and γ_R .

Proof. Let γ be a configuration and let p be a process such that $EBroadcast(\gamma(p))$. We note r the root of the tree which contains p in γ . Lemma 8 applies in γ : $Tree(r)$ is an alive abnormal tree in γ .

Let $\gamma = \gamma_0\gamma_1\dots$ be an execution starting at γ . Let γ_R be the first configuration, if any, in this execution such that p executes an R -action during the step $\gamma_R \mapsto \gamma_{R+1}$. We assume that γ_R exists. Then at some step, $\gamma_i \mapsto \gamma_{i+1}$, p executes a EB -action, with $i < R$.

Lemma 8 applies iteratively at γ_0 and until γ_i : either $Tree(r)$ has disappeared in γ_1 (and so between γ_0 and γ_{i+1}), or p stays in $Tree(r)$ in γ_1 (and so between γ_0 and γ_{i+1}).

If $Tree(r)$ has not yet disappeared in γ_{i+1} , $p \in Tree(r)$ in γ_{i+1} with $\gamma_{i+1}(p).status = EB$. Here, Lemma 7 applies and proves that $Tree(r)$ has disappeared in γ_R or p is in $Tree(r)$ in γ_R . \square

Lemma 9. *Let p be a process. Let s be a segment of execution. Between any two executions of J -action by p in s , p can only execute J -actions.*

Proof. Let $s = \gamma_0\gamma_1\dots$ be a segment of execution and $p \in V$. Consider two executions of J -action by p during s : one in $\gamma_i \mapsto \gamma_{i+1}$ and the other in $\gamma_j \mapsto \gamma_{j+1}$, with $i < j$. Assume by contradiction that p executes an action different from J -action between γ_{i+1} and γ_j . Let $\gamma_k \mapsto \gamma_{k+1}$ be the first step between γ_{i+1} and γ_j during which p executes some other action: this is a EB -action. Let $\gamma_l \mapsto \gamma_{l+1}$ be the last step between γ_{i+1} and γ_j during which p executes some other action: this is a R -action (hence $k < l$).

Now, Lemma 1 applies since in γ_k , $EBroadcast(p)$ holds, and in some step later $\gamma_l \mapsto \gamma_{l+1}$, p executes a R -action. This proves that in γ_k , $Tree(r)$ is an alive abnormal tree and that in γ_l , $Tree(r)$ is dead or has disappeared. Hence γ_k and γ_l are not in the same segment, a contradiction. \square

Lemma 10. *In a segment of execution, there are at most $(n-1)(n-2)/2$ executions of J -action.*

Proof. Let $p \in V$. First, p only executes J -actions between two J -actions in the same segment (Lemma 9). So, using the guard of J -action, it follows that the value of the $p.idR$ always decreases during any sequence of J -actions, which means that p cannot set $p.idR$ two times to the same value during the segment.

Let A be the set of processes q such that $q.status = C$ at the beginning of the segment. Let B the set of processes q such that q executes an R -action in the segment. $A \cap B = \emptyset$. Indeed, pick a process $q \in A \cap B$. q switches from status C at the beginning to status EB , and then to status EF since some step later, it executes R -action. Hence, there exists a configuration γ_b in the segment such that $EBroadcast(q)$ is true and another γ_r , later on such that R -action occurs: hence Corollary 1 applies and proves that the tree of q in γ_b is abnormal alive and that it dies or disappears some step before γ_r . This contradicts the definition of segment. Hence, $|A| + |B| \leq n$.

Now, $p.idR$ can only get values from the idR of processes in A or from the ID of processes in B . Let $f : V \mapsto \mathbb{N}$ such that $\forall p \in A \cup B$, if $p \in A$, $f(p) = x$, where x is the value of $p.idR$ at the beginning of the segment; otherwise, $f(p) = p$. Let p_0, \dots, p_{k-1} (with $k \leq n$) be the set of processes in $A \cup B$ in ascending order of f . p_i changes at most i times of idR . Hence, in a given segment, the number of executed J -actions, noted $\sharp J$ -action satisfies the following inequality:

$$\sharp J\text{-action} \leq \sum_{i=0}^{k-1} i \leq \sum_{i=0}^{n-1} i = \frac{(n-1)(n-2)}{2}$$

□

By Lemmas 4 and 10, in any execution, there are at most $n+1$ segments, where processes execute at most $(n-1)(n-2)/2$ J -actions. Moreover, by definition, there are at most n steps outside segments (more precisely, the steps where at least one abnormal tree dies or disappears).

Corollary 2. *In any execution, there are at most $\frac{n^3}{2} - n^2 + \frac{n}{2} + 1$ steps containing J -actions.*

Other Actions. Below, we show an upper bound on the number of executions of other actions.

Lemma 11. *In any execution, each process can execute at most n R -actions.*

Proof. Let p be a process. We show, by induction on the number of times, $i \geq 0$, p executes R -action that there are at most $n-i$ abnormal alive trees in the system after p executes i R -actions.

The base case $i=0$ is trivial. Assume now that the property holds at some $i \geq 0$ and consider that p has executed R -action at least $i+1$ times.

If $i=0$ then, using Observation 2, $Tree(p)$ exists and is dead at the configuration ρ_1 when the first R -action occurs. As no alive abnormal tree can appear (Lemma 3), $Tree(p)$ will never be an alive abnormal tree forever. Hence there are at most $n-1$ abnormal alive trees after ρ_1 .

If $i > 0$ then, after the i -th R -action, the status of p is C . Between the i -th and the $(i+1)$ -th R -action, the status of p thus switches from C to EB to EF to C . Hence, meanwhile there exists a configuration ρ_b such that $EBroadcast(q)$ is true and another ρ_r , later on such that the $(i+1)$ -th R -action occurs: Lemma 7 applies and proves that the tree of q in ρ_b , say $Tree(r)$ for some r , is abnormal alive and that it dies or disappears some step before ρ_r . Again, as no alive abnormal tree can appear (Lemma 3), $Tree(r)$ is different from the tree already eliminated in previous induction steps and will never come back to alive abnormal tree, forever. Hence there are at most $n-i-1$ abnormal alive trees after ρ_r .

As by definition, there are at most n abnormal alive trees in the initial configuration, we can conclude that p executes R -action at most n times. □

Let p be a process. p necessarily executes R -action between two executions of EF -action (resp. EB -action). Hence, we have the following corollary.

Corollary 3. *In any execution, a process can execute EB -action and EF -action at most n times, each.*

By Corollaries 2, 3, and Lemma 11:

Theorem 1 (Convergence). Every execution contains at most $\frac{n^3}{2} + 2n^2 + \frac{n}{2} + 1$ steps.

4.2.2 Terminal Configurations

We now show that in a terminal configuration, there is one and only one leader process, known by all processes, *i.e.*, for every two processes, p and q , we have $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process.

Lemma 12. *In a terminal configuration, every process has status C .*

Proof. By contradiction, consider a terminal configuration γ where some process p satisfies $p.status \neq C$. Then two cases are possible:

1. $p.status = EB$. By Observation 1, $\exists q \in V$ such that $q.status = EB \wedge (\forall q' \in RealChildren_q, q'.status \neq EB) \wedge p \in KPath(q)$ (q is the latest process of status EB in the S -Trace). If $RealChildren_q = \emptyset$, then q can execute EF -action. Otherwise, there are two cases. Either $\forall q' \in RealChildren_q, q'.status = EF$ and q can execute EF -action, or $\exists q' \in RealChildren_q, q'.status = C$ then q' can execute EB -action. Hence, in both cases, γ is not terminal, a contradiction.
2. $p.status = EF$. By Observation 1, $\exists q \in V$ such that $q.status = EF \wedge (Root(q) \vee (KinshipOk(q, q.par) \wedge q.par.status \neq EF)) \wedge q \in KPath(p)$ (q is the first process of status EF in the S -Trace).

If $Root(q)$, then $AbRoot(q) \vee SelfRoot(q)$. Now, $q.status = EF$ implies that $AbRoot(q)$ holds. So, in any cases, R -action is enabled at q .

Otherwise, there are two cases. Either $p.par.status = C$, $AbRoot(q)$ holds, and R -action is enabled at q , or $p.par.status = EB$ and using the same argument as in case 1, we can deduce that some process is enabled.

Hence, any cases, γ is not terminal, a contradiction. □

Theorem 2 (Correctness). In a terminal configuration, $\forall p, q \in V, Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process.

Proof. Let γ a terminal configuration. Assume first, by contradiction, that there are at least two leaders. Then, G being connected, $\exists p, q \in V$ such that $Leader(\gamma(p)) \neq Leader(\gamma(q))$ and $q \in \mathcal{N}_p$. Assume without loss of generality that $Leader(\gamma(p)) = \gamma(p).idR < \gamma(q).idR = Leader(\gamma(q))$. By Lemma 12, $p.status = q.status = C$. Then, either $EBroadcast(q)$ is true and q can execute EB -action or q can execute J -action. Hence γ is not terminal, a contradiction.

Assume now that the leader is not one of the processes, *i.e.*, is a fake ID. Let $p \in V$ such that its level is minimum. Notice that $\gamma(p).status = C$ by Lemma 12. If $SelfRoot(p)$ holds in γ , $\gamma(p).idR \neq p$. So, $AbRoot(p)$ holds and p can execute EB -action. Otherwise, there is $q \in \mathcal{N}_p$ such that $\gamma(p).par = q$. The level of p being minimum, we have $\gamma(p).level \leq \gamma(q).level$. So, $AbRoot(p)$ holds and p can execute EB -action. Hence, γ is not terminal, a contradiction. □

Using Theorem 2, there is exactly one root in a terminal configuration (the leader elected). So the graph of kinship relations in a terminal configuration contains exactly one tree. Hence, we can conclude:

Remark 2. In a terminal configuration, G_{kr} is a spanning tree rooted at the leader.

Theorems 1 and 2 establish the self-stabilization, silence, and step complexity of Algorithm \mathcal{LE} . Moreover, note that idR and $level$ can be stored in $\Theta(\log n)$ bits. Hence, we can conclude:

Theorem 3. Algorithm \mathcal{LE} is a self-stabilizing and silent leader election algorithm working under a distributed unfair daemon. Its step complexity is at most $\frac{n^3}{2} + 2n^2 + \frac{n}{2} + 1$ steps. Its memory requirement is $\Theta(\log n)$ bits per process.

4.3 Complexity Analysis

In this section, we study the complexity of Algorithm \mathcal{LE} in rounds.

4.3.1 Stabilization Time in Rounds

Clean Configurations. First, we study the “good” cases, *i.e.*, when the system is in a clean configuration (defined below). From such configurations, the execution consists in building a tree rooted at ℓ using J -action only. Once, the tree is built, the system is in a terminal configuration, where every process has elected ℓ .

Definition 12 (Clean configuration). A configuration γ is called a *clean configuration* if and only if for every process p , $\neg EBroadcast(p) \wedge p.status = C$ holds in γ . A configuration that is not clean is said to be *dirty*.

Remark 3. By definition, in a clean configuration, every process p has status C and either p is a normal root, *i.e.*, $SelfRoot(p) \wedge SelfRootOk(p)$ or (exclusively) $KinshipOk(p, p.par)$ holds.

Remark 4. Notice that in a clean configuration, the only action a process p can execute is J -action, provided that $Join(p)$ holds. Note also that $Allowed(p)$ always holds due to Remark 3. Verifying $Join(p)$ then reduces to: $\exists q \in \mathcal{N}_p, (q.idR < p.idR)$. In this case, the value of $p.idR$ can only decrease.

Lemmas 13 to 16 proves that, starting from a clean configuration, the system reaches in $O(\mathcal{D})$ rounds a terminal configuration (see Theorem 4). We first show the set of clean configurations is closed.

Lemma 13. *The set of clean configurations is closed.*

Proof. Let $\gamma \mapsto \gamma'$ be a step such that γ is a clean configuration. By definition, all processes have status C in γ . So, processes can only execute J -action (Remark 4) in $\gamma \mapsto \gamma'$, and consequently all processes have status C in γ' . Now, $\forall p \in V, \neg EBroadcast(p) \wedge p.status = C$ in γ implies that there is no alive abnormal root in γ . By Lemma 2, there is no alive abnormal root in γ' too. Now, the fact that all processes have status C and there is no alive abnormal root in γ' implies that $\forall p \in V, \neg EBroadcast(p) \wedge p.status = C$ in γ' , *i.e.*, γ' is clean. \square

Using Lemma 13, we show below that if a process is enabled in a clean configuration — for the only action it can execute, *i.e.*, J -action — it remains enabled until it executes it.

Lemma 14. *In a clean configuration, if J -action is enabled at p , it remains enabled until it is executed by p .*

Proof. Let $\gamma \mapsto \gamma'$ be a step such that γ is a clean configuration. Assume by contradiction that J -action is enabled at p in γ and not in γ' , but p did not execute J -action between γ and γ' . By Lemma 13, γ' is also a clean configuration. So, $\neg EBroadcast(\gamma'(p)) \wedge \gamma'(p).status = C$ holds.

But $Join(p)$ must be false in γ' . Using Remark 4, this means that there necessarily exists a neighbor of p , say q , such that $\gamma(q).idR < \gamma(p).idR$ but $\gamma'(q).idR \geq \gamma'(p).idR = \gamma(p).idR$. This contradicts Remark 4. \square

Lemma 15. *There is no (fake) idR smaller than ℓ in a clean configuration.*

Proof. Let γ be a clean configuration. Assume there exists a process of idR smaller than ℓ . Let p be such a process such that $p.idR$ is minimum among all the processes and $p.level$ is minimum among all the processes having idR minimum.

Note that $p.idR \neq p$ and consequently $SelfRootOk(p)$ is false in γ . Hence (Remark 3), $KinshipOk(p, p.par)$ holds in γ . Since we take p of minimum idR , $p.idR \leq p.par.idR$ in γ . As $GoodIdR(p, p.par)$ implies that $p.idR \geq p.par.idR$, $p.idR = p.par.idR$. Now, $GoodLevel(p, p.par)$ implies that $p.level = p.par.level + 1$, which contradicts the minimality of $p.level$. \square

For any process p , p can only set $p.idR$ to its own ID or copy the value of $q.idR$, where q is one of its neighbors. So, we have the following remark:

Remark 5. No fake ID is created during any step.

Lemma 16. *In a clean configuration, if the idR of a process p is ℓ , p is disable forever.*

Proof. Let γ be a clean configuration. Let p be a process with $\gamma(p).idR = \ell$. By Remark 4, only J -action can be enabled in γ and its guard reduces to $\exists q \in \mathcal{N}_p, (q.idR < p.idR)$. But Lemma 15 ensures that this can not be true, hence p is disabled in γ . Then, by Lemma 13 and Remark 5, this will be true forever. \square

Corollary 4. *A clean configuration where $\forall p \in V, p.idR = \ell$, is terminal.*

Theorem 4. In a clean configuration, the system reaches a terminal configuration where $\forall p \in V, p.idR = \ell$ in at most \mathcal{D} rounds.

Proof. Consider any execution e that starts from a clean configuration. In the following, we denote by ρ_i the first configuration of the i th round in e . We show by induction on the distance $d \geq 0$ between the processes and ℓ that $\forall p \in V$ such that $\|p, \ell\| \leq d$, $\rho_d(p).idR = \ell$.

Base case: If $\|p, \ell\| = 0$, $p = \ell$. Note that $KinshipOk(p, p.par)$ cannot hold in ρ_0 since $GoodIdR(p, p.par)$ would implies that $p.idR < p$ which is false by Lemma 15. Hence, from Remark 3, $SelfRoot(p) \wedge SelfRootOk(p)$ holds in ρ_0 and $\rho_0(p).idR = p = \ell$.

Induction step: Assume the property holds at some $d \geq 0$. If $\|p, \ell\| = d + 1$, $\exists q \in \mathcal{N}_p$ such that $\|q, \ell\| = d$. By induction hypothesis and by Lemma 16, $q.idR = \ell$ and q is disabled forever since ρ_d . If $p.idR = \ell$ in ρ_d , it remains so forever (Lemma 16). If $p.idR \neq \ell$ in ρ_d then $q.idR < p.idR$ (Lemma 15). Then, J -action is enabled at p in ρ_d and remains enabled until p executes it (Lemma 14). As there is no fake ID smaller than ℓ (Lemma 15), $p.idR = \ell$ after p executes J -action, *i.e.*, after at most one round. Hence, $p.idR = \ell$ in ρ_{d+1} .

As $\mathcal{D} \geq \max\{\|p, \ell\|, p \in V\}$, in at most \mathcal{D} rounds, the system reaches a configuration where $\forall p \in V, p.idR = \ell$. By Corollary 4, this configuration is terminal. \square

Dirty Configurations. In the previous section, we showed that, if the initial configuration is clean, the system reaches a terminal configuration in at most \mathcal{D} rounds. But what happens if the initial configuration is dirty, *i.e.*, if there is a process p such that $EBroadcast(p)$ holds or $p.status \neq C$. In this section, we prove that starting from a dirty configuration, the system reaches a clean configuration in at most $3n$ rounds. More precisely, we show that a dirty configuration contains abnormal trees that are “cleaned” in at most $3n$ rounds. The system will be in a clean configuration afterwards.

Lemma 17. *In an dirty configuration, there exists at least one abnormal root.*

Proof. Let γ be a dirty configuration. Then $\exists p \in V$ such that $p.status \neq C \vee EBroadcast(p)$. We search for an abnormal root.

1. If $p.status \in \{EB, EF\}$, using Observation 1, there is $q \in KPath(p)$ such that $q.status \in \{EB, EF\} \wedge Root(q)$. Then, $AbRoot(q) \vee SelfRoot(q)$. Now, $SelfRoot(q) \wedge q.status \in \{EB, EF\}$ implies $AbRoot(q)$. Hence, in all cases, $AbRoot(q)$ holds.
2. If $EBroadcast(p)$ holds, Lemma 8 applies and we are done.

\square

We have just shown that there are abnormal roots (and so abnormal trees) in dirty configurations. Below, we prove that these abnormal trees will disappear after three waves of “cleaning”. After the first wave, an abnormal tree becomes dead (Theorem 5), after the second wave the root gets the status EF (Theorem 6) and finally after the third wave there is no more abnormal trees (Theorem 7), hence the system is in a clean configuration.

The following technical lemma is used in the proof of Theorem 5.

Lemma 18. *When EB -action is enabled at a process p , it remains enabled until p executes EB -action.*

Proof. Assume that EB -action is enabled at a process p in a configuration γ , but p did not execute EB -action during the step $\gamma \mapsto \gamma'$. Notice that p does not execute any action during this step, as guards are mutually exclusive. As EB -action is enabled in γ , $\gamma(p).status = C$ and then, $\gamma'(p).status = C$.

First, assume $AbRoot(p)$ holds in γ . If $SelfRoot(p) \wedge \neg SelfRootOk(p)$ holds in γ and, as these predicates only depends on the local state of p and as p does not execute any action during the step, it also holds in γ' : the action is still enabled in γ' . Otherwise, $\neg SelfRoot(p) \wedge \neg KinshipOk(p, p.par)$ holds in γ . These predicates only depends on the local state of p and its parent. Now, $Allowed(p.par)$ does not hold in γ because of p , so $p.par$ cannot execute R -action nor J -action during $\gamma \mapsto \gamma'$. Then,

either $p.par$ executes EF -action, changes its status to EF and $GoodStatus(p, p.par)$ is false in γ' , or it executes EB -action and changes its status to EB . In these two cases, $EBroadcast(p)$ holds in γ' .

Now, assume $p.par.status = EB$, $p.par$ can only execute EF -action and change its status to EF . Then, $GoodStatus(p, p.par)$ is false in γ' , which implies that $EBroadcast(p)$ holds in γ' . \square

Theorem 5. In at most n rounds, the system reaches a configuration where every abnormal tree (if any) is dead.

Proof. Consider any execution e . We denote by γ_{R_i} the first configuration of the i th round in e . We show by induction on the length of the $KPaths$ that, in at most d rounds ($d \geq 1$), $\forall p \in V$, if p is in an abnormal tree and $|KPath(p)| \leq d$ then p is dead.

Base case: If p is in an abnormal tree and $|KPath(p)| = 1$, p is an abnormal root. As no alive abnormal root is created (Lemma 2), if p is alive, it is an alive abnormal root since γ_{R_1} and if predicate $(p.status = C \wedge AbRoot(p))$ becomes false, then it remains false forever. Hence, it is sufficient to show that any alive abnormal root is no more an alive abnormal root after one round.

By definition, EB -action is enabled at p in γ_{R_1} and p executes EB -action in the first round (Lemma 18). Hence, p is dead at the end of the first round, and we are done.

Induction Step: We first show that for every $p \in V$, if $(p.status = C \wedge |KPath(p)| \leq d + 1)$ is false in some configuration after d rounds then, this remains false forever.

Assume by contradiction that the predicate “ $p.status = C \wedge |KPath(p)| \leq d + 1$ ” is false in γ_i , but true in γ_{i+1} . By induction hypothesis, $|KPath(p)| = d + 1 > 1$ in γ_{i+1} (indeed, p is alive in γ_{i+1}). So, $\gamma_{i+1}(p).par \neq p$. So, let $q \in \mathcal{N}_p$ such that $\gamma_{i+1}(p).par = q$. By definition, $|KPath(q)| = d$ in γ_{i+1} . By induction hypothesis, $\gamma_{i+1}(q).status \in \{EB, EF\}$. Now, $p.status = C$ and $|KPath(p)| > 1$ in γ_{i+1} , so p is not an abnormal root in γ_{i+1} . Hence, $\gamma_{i+1}(q).status = EB$ (by Observation 1) and, consequently, $\gamma_i(q).status \in \{C, EB\}$.

- If $\gamma_i(q).status = EB$, then p does not execute any action in the step $\gamma_i \mapsto \gamma_{i+1}$ (otherwise, $\gamma_{i+1}(p).status \neq C$ or $\gamma_{i+1}(p).par \neq q$). Moreover, q does not execute any action during this step (otherwise its status would be different from EB in γ_{i+1}). Hence, the predicate “ $p.status = C \wedge |KPath(p)| \leq d + 1$ ” has the same value in γ_i and γ_{i+1} , a contradiction.
- If $\gamma_i(p).status = C$, then q is in an alive abnormal tree in γ_i (indeed, q executes EB -action in $\gamma_i \mapsto \gamma_{i+1}$, and so Lemma 8 applies). As q is alive in γ_i , we have $|KPath(q)| > d$ in γ_i by induction hypothesis. Moreover, q is not an abnormal root (there is no more alive abnormal root after the first round, see the base case). Hence, the status of its parent in γ_i is EB . Now, $|KPath(q)| > d$ and $S-Trace(KPath(q)) = EB^+C$ in γ_i . So, q is the only one in its $KPath$ that executes an action in $\gamma_i \mapsto \gamma_{i+1}$ and this action is EB -action, which maintains the $KinshipOk$ relation. Hence, $|KPath(q)| > d$ in γ_{i+1} and consequently, $|KPath(p)| > d + 1$ in γ_{i+1} , a contradiction.

Hence, for every process p , if $(p.status = C \wedge |KPath(p)| \leq d + 1)$ is false in some configuration after d rounds, $(p.status = C \wedge |KPath(p)| \leq d + 1)$ remains false forever.

Now, EB -action is continuously enabled $\forall p$ such that p is alive $|KPath(p)| = d + 1$ in $\gamma_{R_{d+1}}$ (by induction hypothesis and Lemma 18). So, p becomes dead during the round and, after that round, there will be no alive process p such that $|KPath(p)| \leq d + 1$ forever.

$n \geq \max\{|KPath(p)|, \forall p \in V\}$. Hence, any process in an abnormal tree becomes dead in at most n rounds, and we are done. \square

Lemma 19. If EF -action is enabled at a process p , it remains enabled until p executes EF -action.

Proof. Assume by contradiction EF -action is enabled at a process p in configuration γ and is not enabled in the next configuration γ' , but p did not execute EF -action during the step $\gamma \mapsto \gamma'$. Notice that p does not execute any action during this step, as guards are mutually exclusive. As $EFfeedback(p)$ holds in γ , $\gamma(p).status = \gamma'(p).status = EB$. As $EFfeedback(p)$ does not hold in γ' and no process can execute

J -action and choose a process of status EB as parent, $\exists q \in RealChildren_p$ such that $\gamma(q).status = EF$ and $\gamma'(q).status \neq EF$. Now, because $\gamma(q).status = EF$, q can only execute R -action. However, as $q \in RealChildren_p$, $KinshipOk(q, p)$ holds in γ and then q is not a root. So, q cannot execute any action and change its status during $\gamma \mapsto \gamma'$, a contradiction. \square

Theorem 6. Let γ be a configuration containing abnormal trees and where all abnormal trees are dead. In at most n rounds from γ , the system reaches a configuration where the status of all abnormal roots is EF .

Proof. Consider any execution e starting from a configuration that contains abnormal trees and where all abnormal trees are dead. We denote by γ_{R_i} the first configuration of the i th round.

Claim 1: $\forall p \in V, \forall i \geq R_1$, if $\gamma_i(p).status \neq C$, then $\forall j \geq i, \gamma_j(p).status \neq EB$.

Assume by contradiction that $\gamma_j(p).status \neq EB$ and $\gamma_{j+1}(p).status = EB$, with $\gamma_j \mapsto \gamma_{j+1}$. Then, $p.status = C$ in γ_j and EB -action is enabled at p in γ_j . So, p is in an alive abnormal tree in γ_j (Lemma 8), a contradiction with Lemma 3.

Let $MaxKPath(p)$ be the maximal $KPath$ that contains p in the current configuration. Let $L(p) = |MaxKPath(p)| - |KPath(p)|$. Let $EBL(p, k) \equiv p.status = EB \wedge L(p) = k$.

Claim 2: $\forall i \geq R_1$, if $EBL(p, k_i)$ holds in γ_i , then $\forall j > i, \forall k_j < k_i, \neg EBL(p, k_j)$ holds in γ_j .

The case $k_i = 0$ is direct. Assume $k_i > 0$. Assume by contradiction that $EBL(p, k_i)$ holds in γ_i and $EBL(p, k_j)$ holds in γ_j with $j > i$ and $k_j < k_i$. Hence, $\gamma_i(p).status = \gamma_j(p).status = EB$. There are two cases:

- $p.status = EB$ in all the configurations between γ_i and γ_j . Consider the step $\gamma_i \mapsto \gamma_{i+1}$. Let q such that $p \in KPath(q)$. So, $KPath(q) = q_0 \dots q_i = p \dots q_k = q$ and $S-Trace(KPath(q)) = EB^+EF^*$. There is a unique process in $KPath(q)$ that can execute an action (the only one of status EB with children of status EF). If it executes an action, it is EF -action which maintains $KinshipOk$ relation. Hence, $\forall q' \in KPath(q)$ in $\gamma_i, q' \in KPath(q)$ in γ_{i+1} . So, $EBL(p, k_{i+1})$ holds with $k_{i+1} \geq k_i$. With the same argument on step $\gamma_{i+1} \mapsto \gamma_{i+2}$, etc., until step $\gamma_{j-1} \mapsto \gamma_j$, we show that $EBL(p, k_j)$ is true in γ_j with $k_j \geq k_i$, a contradiction.
- There is a configuration between γ_i and γ_j where $p.status \neq EB$. So, $\exists x, i < x < j$, such that $\gamma_x(p).status \neq EB$ and $\gamma_{x+1}(p).status = EB$. With the same argument as in Claim 1, we get a contradiction.

We show by induction on $L(p)$ that in γ_{R_d} , $\nexists p \in V$ such that $EBL(p, i)$ holds for $i \leq d - 2$, with $d \geq 2$.

Base case: There are three cases:

1. If $L(p) = 0$ in γ_{R_1} and $\gamma_{R_1}(p).status = EB$, then EF -action is enabled at p in γ_{R_1} , p executes EF -action in the first round by Lemma 19 and p gets status EF . By Claim 1, $p.status$ remains different from EB forever and $EBL(p, 0)$ is false in γ_{R_1} .
2. If $\gamma_{R_1}(p).status \neq EB$, $p.status \neq EB$ forever (Claim 1) and then $EBL(p, 0)$ is false forever.
3. If $EBL(p, k)$ holds in γ_{R_1} with $k > 0$, $EBL(p, 0)$ is false forever (Claim 2).

Induction step: There are four cases:

1. If $L(p) = d + 1$ and $\gamma_{R_{d+2}}(p).status = EB$, $\forall q \in RealChildren_p$ in $\gamma_{R_{d+2}}$, $L(q) < d + 1$ by definition and $\gamma_{R_{d+2}}(q).status \neq EB$ by induction hypothesis. Now, the tree is dead, so $\gamma_{R_{d+2}}(q).status = EF$. Hence, EF -action is enabled at p in $\gamma_{R_{d+2}}$, p executes EF -action in the round (Lemma 19) and gets status EF . By Claim 1, $p.status \neq EB$ forever so $EBL(p, d + 1)$ is false at the end of the $d + 2$ th round and remains false forever.
2. If $L(p) = d + 1$ and $\gamma_{R_{d+2}}(p).status \neq EB$, then $p.status \neq EB$ forever and $EBL(p, d + 1)$ is false forever.
3. If $L(p) < d + 1$, by induction hypothesis $\gamma_{R_{d+2}}(p).status \neq EB$ and p is in case 2.

4. If $EBL(p, k)$ holds in $\gamma_{R_{d+2}}$ with $k > d + 1$, $EBL(p, i)$ is false forever for $i \leq d + 1$ (Claim 2).

$n \geq \max\{L(p) \mid p \in V\}$. Hence, in at most n rounds, there is no more process of status EB in the abnormal trees, which are dead. So, all processes (and in particular the abnormal roots) in the abnormal trees have status EF . □

Lemma 20. *If all abnormal trees are dead and R -action is enabled at a process p , then R -action remains enabled at p until p executes it.*

Proof. Let γ be a configuration, where all abnormal trees are dead. Assume, by contradiction, that R -action is enabled at a process p in a configuration γ and is not enabled in the next configuration γ' , but p did not execute R -action during the step $\gamma \mapsto \gamma'$. Notice that p does not execute any action during this step, as guards are mutually exclusive.

As R -action is enabled in γ and p does not execute an action during the step, $\gamma(p).status = \gamma'(p).status = EF$.

If $SelfRoot(p) \wedge \neg SelfRootOk(p)$ holds in γ , it also holds in γ' because p does not execute an action between γ and γ' and these predicates only depends on the local state of p .

Otherwise $\neg SelfRoot(p) \wedge \neg KinshipOk(p, p.par)$ holds in γ . Let $q = p.par$. If q does not execute an action between γ and γ' , p is still an abnormal root. Otherwise, three cases are possible:

- $\neg GoodIdR(p, q)$ holds in γ . First, if $\gamma(p).idR < \gamma(q).idR$. If q executes EB -action or EF -action during the step, the idR of q does not change, so $\gamma'(p).idR < \gamma'(q).idR$, and $AbRoot(p)$ holds in γ' . Otherwise q executes R -action or J -action. Then $\gamma'(q).status = C$, so $\neg GoodStatus(p, q)$ and $AbRoot(p)$ holds in γ' . If $\gamma(p).idR \geq p$, the idR is not modified during the step, so $\gamma'(p).idR = \gamma(p).idR \geq p$ and $AbRoot(p)$ holds in γ' .
- $\neg GoodLevel(p, q)$ holds in γ . Then $\gamma(p).idR = \gamma(q).idR$ but $\gamma(p).level \neq \gamma(q).level + 1$. If q executes EB -action or EF -action, its idR and its $level$ do not change, so $\gamma'(p).idR = \gamma'(q).idR$ and $\gamma'(p).level \neq \gamma'(q).level + 1$, so $AbRoot(p)$ holds in γ' . Otherwise, q executes R -action or J -action. Then $\gamma'(q).status = C$, so $\neg GoodStatus(p, q)$ and $AbRoot(p)$ holds in γ' .
- $\neg GoodStatus(p, q)$ holds in γ . Then $\gamma(q).status = C$, and q can only execute EB -action or J -action between γ and γ' . If q executes EB -action then $EBroadcast(q)$ holds in γ , so q is in an abnormal tree (Lemma 8). But, by hypothesis, all abnormal trees are dead in γ , so $\gamma(q).status \neq C$, a contradiction. If q executes J -action then $\gamma'(q).status = C$, so $\neg GoodStatus(p, q)$ and $AbRoot(p)$ holds in γ' .

Thus, $\gamma'(p).status = EF$ and $AbRoot(p)$ holds in γ' and, consequently, $Allowed(p)$ is false in γ' . So $\exists q \in \mathcal{N}_p$ such that $q \in Children_p \wedge \neg KinshipOk(q, p)$ holds in γ' but $\gamma'(q).status = C$. Two cases are possible:

- If $q \notin Children_p$ in γ , then q executes J -action during the step $\gamma \mapsto \gamma'$ and $Min_q = p$. But $\gamma(p).status = EF$, a contradiction.
- Otherwise $q \in Children_p$ in γ and $\gamma(q).status \neq C$. q executes either EF -action and $\gamma'(q).status = EF$ or R -action and $\gamma'(q).par \neq p$, so $q \notin Children_p$ in γ' , a contradiction. □

Definition 13 (Abnormal process). A process p is called *abnormal* process if and only if p belongs to an abnormal tree. p is said to be *normal*, otherwise.

As no process can join a dead abnormal tree (Remark 1) and no alive abnormal tree can be created (Lemma 3), we have the following remark:

Remark 6. In a configuration where every abnormal tree is dead, the number of abnormal processes can only decrease.

Theorem 7. Starting from a configuration where every abnormal tree is dead and the status of their roots is EF , there is no more abnormal processes in at most n rounds.

Proof. Let γ_0 be a configuration where all abnormal trees are dead and the status of their roots is EF . By Observation 1, all abnormal processes have status EF in γ_0 . So, from γ_0 , no process can be ever an abnormal process with a status different of EF (such a process can only execute R -action, then it is a normal process forever, by Lemma 3). Then, by definition, the number of abnormal processes in γ_0 is at most n . Moreover, by Remark 6, it is sufficient to show that in any configuration γ_k reachable from γ_0 , if the number of abnormal processes is not null, then at least one of them becomes normal within the next round.

So, let assume that some process p is abnormal in γ_k . Then, $\gamma_k(p).status = EF$. By Observation 1 and Lemma 20, the initial extremity r of $KPath(p)$ is an abnormal process (of status EF) and executes R -action within the next round. After executing R -action, r is normal (actually, r becomes a self root), and we are done. □

By definition, the root of a normal tree has the status C . So, by Observation 1, we have:

Remark 7. Every process has the status C in a configuration containing no abnormal processes. Moreover, this configuration is clean.

Using Lemma 17 and Theorems 5 to 7, we can conclude:

Theorem 8. In at most $3n$ rounds, the system reaches a clean configuration.

Then, using Theorems 4 and 8 we get:

Theorem 9 (Round Complexity). In at most $3n + \mathcal{D}$ rounds, the system reaches a terminal configuration.

4.3.2 Worst Case Analysis of the Stabilization Time

Lower Bound on the Worst Case Stabilization Time in Rounds. We now show that the bound proposed in Theorem 9 cannot be improved. To see this, we exhibit a construction that gives, $\forall n \geq 4$, $\forall \mathcal{D}, 2 \leq \mathcal{D} \leq n - 2$, a network of n processes whose diameter is \mathcal{D} , from which there is a possible synchronous execution that lasts exactly $3n + \mathcal{D}$ rounds. (Recall that every synchronous execution is possible under the distributed unfair daemon.)

We consider a network $G = (V, E)$ composed of n processes $V = \{p_1, \dots, p_n\}$ such that p_i has ID i , $\forall i \in [1..n]$. Figure 7a shows the system in its initial configuration. In details, processes p_1, p_n, \dots, p_2 form a chain, *i.e.*, $\{p_1, p_n\} \in E$ and $\{p_i, p_{i-1}\} \in E \forall i = 3 \dots n$.

We add k “legs”, with $2 \leq k \leq n - 2$, as follows:

If $k = n - 2$ then $\{p_2, p_1\} \in E$ and $\forall i \in [4..n], \{p_2, p_i\} \in E$,

Otherwise $\forall i \in [4..k + 3], \{p_2, p_i\} \in E$.

Notice that the diameter of the network is $n - k$ and can be adjusted by adding or removing some legs.

We assume the following initial configuration:

- $p_i.idR = 0 \forall i \in [1..n]$,
- $p_1.level = n - 1$ and $p_1.par = p_n$,
- $p_2.par = p_2$ and $p_2.level = 0$,
- $p_i.level = i - 2$ and $p_i.par = p_{i-1}, \forall i \in [3..n]$.

We consider a synchronous daemon, *i.e.* in a configuration γ , every process in $Enabled(\gamma)$ is selected by the daemon to execute an action. So, in this case, every round lasts exactly one round.

The execution is then as follows:

- $p_2, p_3, p_4 \dots p_n, p_1$ sequentially execute EB -action: n rounds. (See Figure 7b.)
- $p_1, p_n, p_{n-1}, \dots, p_2$ sequentially execute EF -action: n rounds. (See Figure 7c.)
- p_2 and p_3 sequentially execute R -action: 2 rounds. (See Figure 7d)

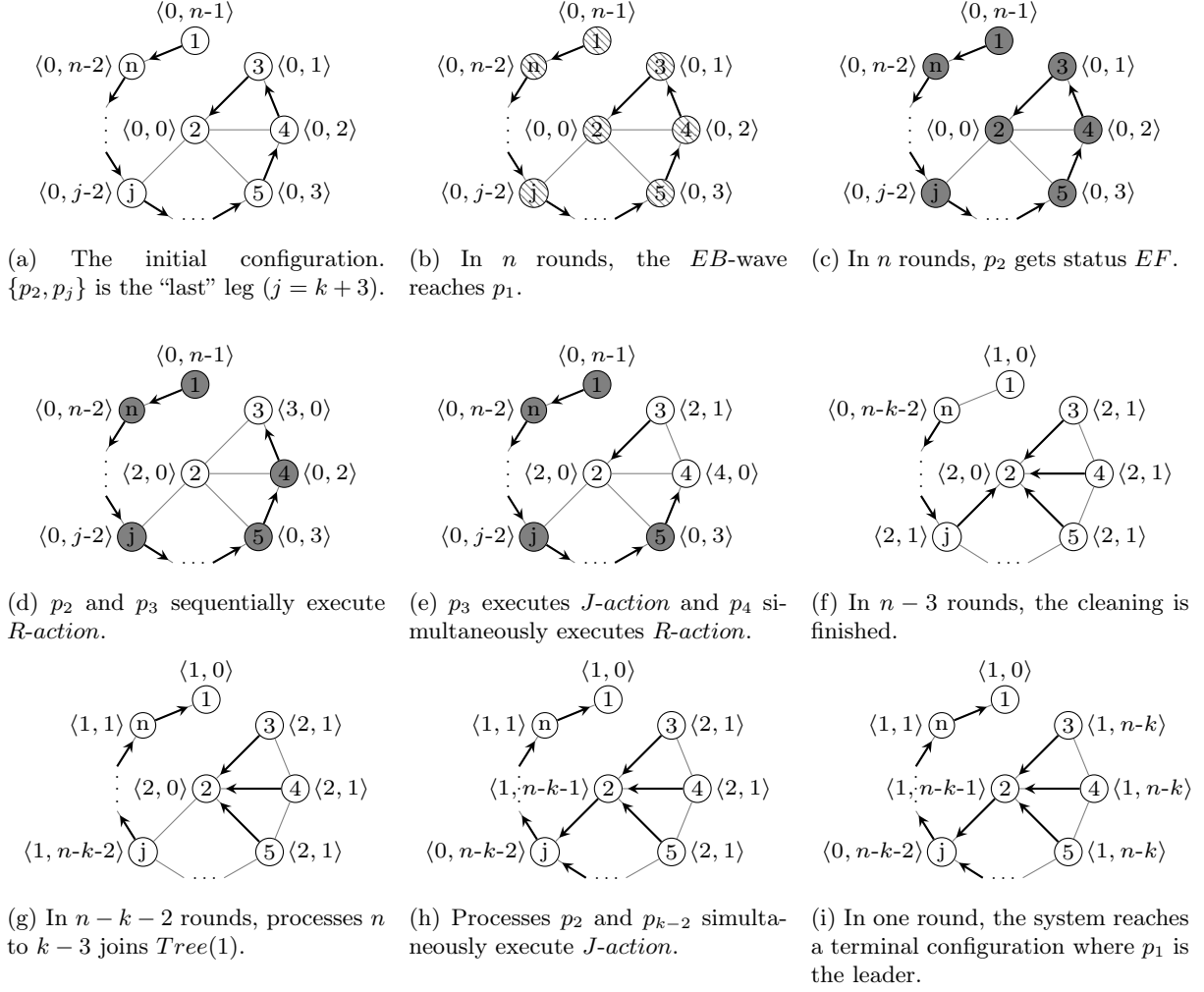


Figure 7: An example in $3n + \mathcal{D}$ rounds

- For $i = 4 \dots n$, simultaneously p_i and p_{i-1} respectively executes R -action and J -action, in particular, p_{i-1} joins $Tree(p_2)$: $n - 3$ rounds. (See Figures 7e and 7f.)
- p_1 executes R -action and p_n executes J -action simultaneously: 1 round.
- For $i = n \dots k + 3$, i executes J -action to join $Tree(1)$: $n - k - 2$ rounds. (See Figure 7g.)
- p_2 and p_{k+2} simultaneously execute J -action to join $Tree(1)$: 1 round. (See Figure 7h.)
- p_3, \dots, p_{k+1} simultaneously execute J -action and then the configuration is terminal: 1 round. (See Fig. 7i.)

Hence, the execution lasts exactly $3n + (n - k) = 3n + \mathcal{D}$ rounds.

Lower Bound on the Worst Case Stabilization Time in Steps We show that the bound given in Theorem 1 can be asymptotically matched, *i.e.*, we give an example of possible execution that stabilizes in $\Omega(n^3)$ steps, for every $n \geq 4$.

We consider a network $G = (V, E)$ composed of n processes $V = \{p_1, \dots, p_n\}$ such that p_i has ID $n + i$, $\forall i \in [1..n]$. Figure 8a shows the network in this initial configuration. In details, there are $2n - 3$ edges: $\{p_i, p_{i+1}\} \forall i = 1 \dots n - 2$ and $\{p_i, p_n\} \forall i = 1 \dots n - 1$. The initial configuration is as follows:

- $p_i.idR = i \forall i \in [1..n - 1]$, and $p_n.idR = 2n$.
- $p_i.par = p_i$, $p_i.level = 0$ and $p_i.status = C \forall i \in [1..n]$.

We consider the following execution:

- For $i = n - 1 \dots 1$ (step -1), we clean $Tree(p_i)$ the following way:

1. For $j = n - 2 \dots i$ (step -1),¹
 - (a) For $k = j + 1 \dots n - 1$ (step +1),²
 - p_k joins $Tree(p_j)$.

This part lasts $\sum_{k=1}^{n-1-i} k$ steps.

2. $p_i, p_{i+1}, \dots, p_{n-1}$ sequentially execute EB -action: $n - i$ steps.
3. $p_{n-1}, p_{n-2}, \dots, p_i$ sequentially execute EF -action: $n - i$ steps.
4. $p_i, p_{i+1}, \dots, p_{n-1}$ sequentially execute R -action: $n - i$ steps.

Figures 8e to 8h show the cleaning of $Tree(p_{n-3})$.

- After all abnormal trees have been cleaned, processes p_{n-1} to p_2 join $Tree(p_1)$ similarly as Case 1: $\sum_{i=1}^{n-2} i$ steps (Figure 8j).
- p_n executes J -action to join $Tree(p_1)$: 1 step (Figure 8k).

Hence, the complete execution lasts:

$$\left(\sum_{i=1}^{n-1} (3(n-i) + \sum_{k=1}^{i-1} k) \right) + \left(\sum_{i=1}^{n-2} i \right) + 1 = \frac{n^3}{6} + \frac{5}{2}n^2 - \frac{11}{3}n + 2\text{steps}$$

5 Step Complexity of [8]

In this section, we study the step complexity of the algorithm given in [8]. In the following, this algorithm is called \mathcal{DLV} .³ Below, we show that its stabilization time is not polynomial in steps.

First, we give the code of algorithm \mathcal{DLV} and an informal explanation of its principle in Subsection 5.1. Then, we give an example of a class of network and possible computation that stabilizes in $\Omega(n^4)$ steps in Subsection 5.2. Finally, in Subsection 5.3, we generalize the previous example to a class of network and possible computation that stabilizes in $\Omega(n^{\alpha+1})$ for any $\alpha \geq 3$.

¹Of course, when $n - 2 < i$, there is no iteration.

²Of course, when $j + 1 > n - 1$, there is no iteration.

³To avoid confusion with the algorithm names in our explanation, we use the name \mathcal{DLV} , that stands for ‘‘Datta, Larmore, Vemula.’’

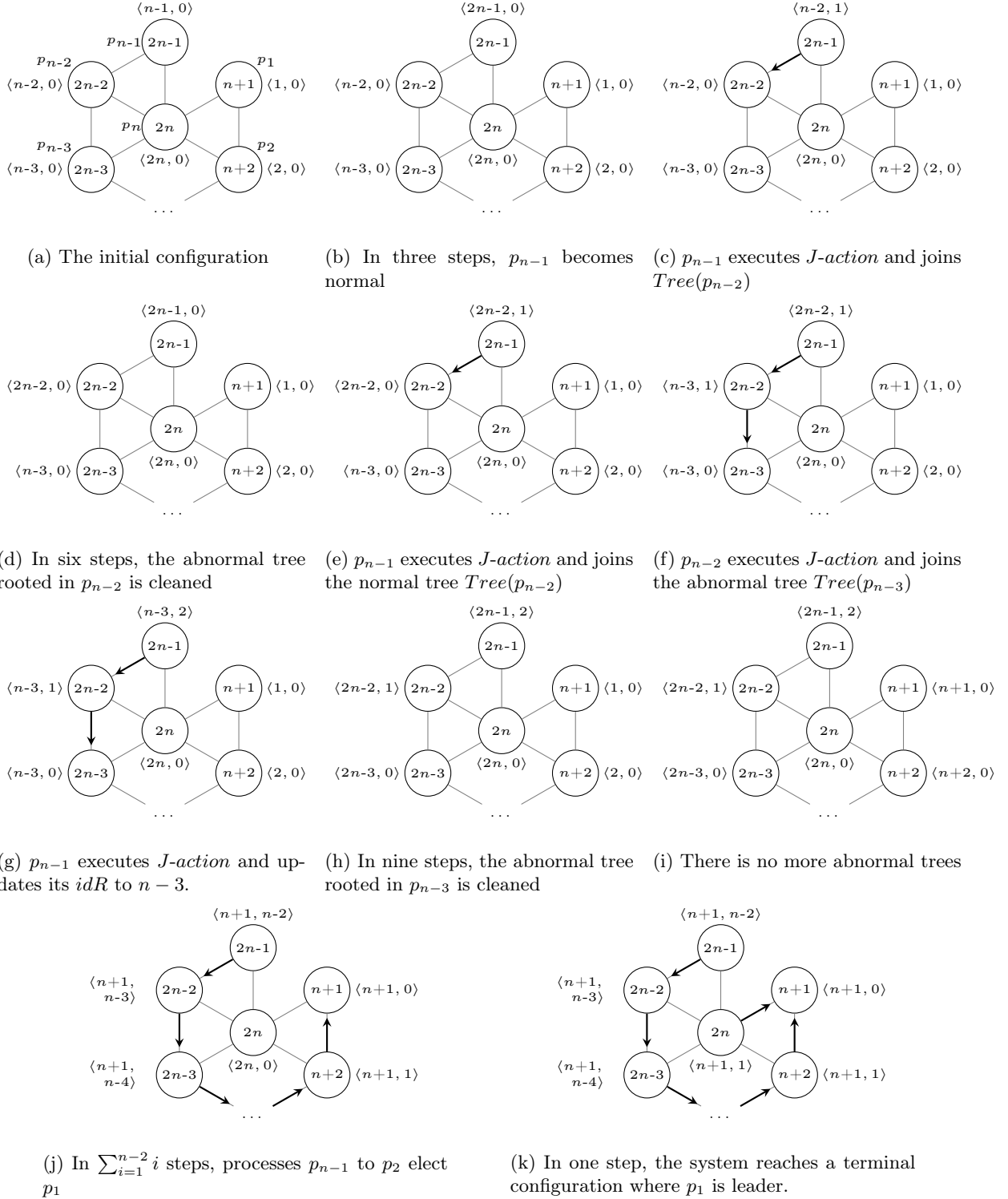


Figure 8: An example in $\Omega(n^3)$ steps

5.1 Informal explanation of \mathcal{DLV}

Algorithm \mathcal{DLV} (refer to Algo. 2) elects the process of minimum ID, ℓ , and builds a minimum spanning tree rooted at ℓ . To ensure that every process knows which one is elected, it maintains a variable *leader* to save its current leader. Variables *parent* and *level* are used to represent the tree. The *key* of a process p is the combination of its two variables $p.\text{leader}$ and $p.\text{level}$. Notice that the keys are ordered by a lexical order.

When a process p has a neighbor with a smaller key, p executes action J , gets the successor key of the smaller such neighbor ($\text{BestNbrKey}(p)$) and chooses this latter as parent. Notice that, contrary to our algorithm, p can execute action J and change its parent if there is a process with the same leader but with a level smaller than $p.\text{level} - 1$, in order to build a minimum spanning tree.

As in \mathcal{LE} , they define a “good relation” between a process p and its parent: $\text{IsTrueChld}(p)$. It ensures that the key of p is the successor key of its parent and that its leader is smaller than its own ID. Then, a maximal set of processes linked by *parent* pointers and satisfying IsTrueChld relation is a tree. The root of this tree can be a *true root* ($\text{IsTrueRoot}(p)$), *i.e.*, the key of p is its self key ($(p, 0)$). In this case, they said that it is a *normal tree*. Otherwise, the root is a *false root* ($\text{IsFalseRoot}(p)$), *i.e.*, neither a true root nor a true child, and they said that it is an *abnormal tree*.

Color waves. The main difference between \mathcal{DLV} and \mathcal{LE} is the way to deal with these abnormal trees. Instead of using a status and a three waves cleaning, \mathcal{DLV} uses color waves. More precisely, each process has a variable *color*, either 1 or 2. A process can only change its parent to a neighbor of color 2 and after executing action J , the process gets color 1.

A process p of color 2 cannot change its color to 1 when it has possible recruits ($\text{Recruits}(p) \neq \emptyset$), *i.e.* there are some neighbors with a bigger key that may choose p as parent later. Furthermore, a process can change its color, executing actions $C1$ or $C2$, if it has the same color than its parent and if all of its true children have the other color.

To add a new level in the tree, the leaves must change their color to 2. A first wave of actions $C1$ is initiated by the parents of the leaves and absorbed by the root. Then, a second wave of actions $C2$ is initiated by the leaves and also absorbed by the root. When the leaves have color 2, their neighbors can join the tree. But the priorities of the actions prevent a false root to change its color and then, only a true root can absorb a color wave.

Therefore, the colors of the processes in an abnormal tree eventually alternate, *i.e.*, the parents and their real children do not have the same color, and no more process can join the tree: the tree is *color locked*. Then, the root eventually resets to a true root executing action R .

Fig. 9 shows an example of execution with the cleaning of an abnormal tree.

Finally, in $O(n)$ rounds, ℓ is elected and a minimum spanning tree rooted at ℓ is built. Notice that the color waves might never end. A mechanism ensure the silence of the algorithm using the Boolean variable *done*. When a process p believes that the construction of the final tree is finished (because it can no more recruits other processes) and all its true children q have set their variables $q.\text{done}$ to **true**, $p.\text{done}$ is set to **true**. Moreover, a true root r cannot change its color if $r.\text{done}$ holds. We said that r is *color frozen*. Then, in $O(\mathcal{D})$ additional rounds, *done* is propagated from the leaves to the root and the system reach a terminal configuration.

5.2 Example in $\Omega(n^4)$ steps

We consider a network made of $n = L \times \beta$ processes with $L = 14$ and $\beta \geq 2$: $p_{(1,1)}, p_{(1,2)}, \dots, p_{(1,\beta)}, p_{(2,1)}, p_{(2,2)}, \dots, p_{(2,\beta)}, \dots, p_{(14,1)}, p_{(14,2)}, \dots, p_{(14,\beta)}$ such that the ID of $p_{(i,j)}$ is $(i - 1)\beta + j, \forall i \in [1 \dots 14], \forall j \in [1 \dots \beta]$. Fig. 10 shows the structure of the network and the initial configuration. In details, the processes form β columns: $\forall i \in [2 \dots 14], \forall j \in [1 \dots \beta], \{p_{(i,j)}, p_{(i-1,j)}\} \in V$. Moreover, there are three complete bipartite graphs: $\forall j \in [1 \dots \beta], \forall j' \in [1 \dots \beta], j' \neq j, \{p_{(8,j)}, p_{(7,j')}\}, \{p_{(12,j)}, p_{(11,j')}\}$ and $\{p_{(14,j)}, p_{(13,j')}\}$ are in V . These bipartite graphs split the network in four layers:

- Layer 1: line 14
- Layer 2: lines 12 and 13
- Layer 3: lines 8 to 11

Algorithm 2 Algorithm \mathcal{DLV} [8] for process p

Variables

$p.leader$ of some ordered ID type
 $p.level \in \mathbb{N}$
 $p.key = \langle p.leader, p.level \rangle$
 $p.parent \in \mathcal{N}_p \cup \{p\}$
 $p.color \in \{1, 2\}$
 $p.done \in \mathbb{B}$

Macros

$SelfKey(p) \equiv \langle p, 0 \rangle$
 $SuccKey(p) \equiv \langle p.leader, p.level + 1 \rangle$
 $BestNbrKey(p) \equiv \min \{q.key \mid (q \in \mathcal{N}_p) \wedge (SuccKey(q) < SelfKey(p)) \wedge (q.color = 2)\}$
 $TrueChldrn(p) \equiv \{q \in \mathcal{N}_p \mid (q.parent = p) \wedge (q.key = SuccKey(p))\}$
 $FalseChldrn(p) \equiv \{q \in \mathcal{N}_p \mid (q.parent = p) \wedge (q.key \neq SuccKey(p))\}$
 $Recruits(p) \equiv \{q \in \mathcal{N}_p \mid q.key > SuccKey(p)\}$

Predicates

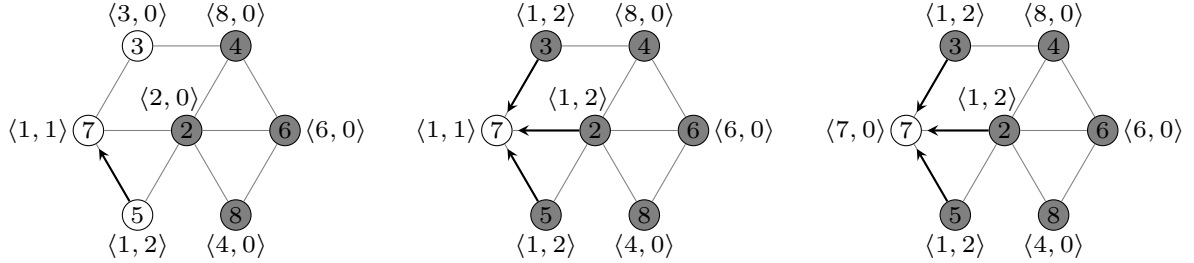
$IsTrueRoot(p) \equiv p.key = SelfKey(p)$
 $IsTrueChld(p) \equiv (p.key = SuccKey(p.parent)) \wedge (p.leader < p)$
 $IsFalseRoot(p) \equiv \neg IsTrueRoot(p) \wedge \neg IsTrueChld(p)$
 $Done(p) \equiv (Recruits(p) = \emptyset) \wedge (\forall q \in TrueChldrn(p), q.done)$
 $ColorFrozen(p) \equiv IsTrueRoot(p) \wedge p.done$

Guards

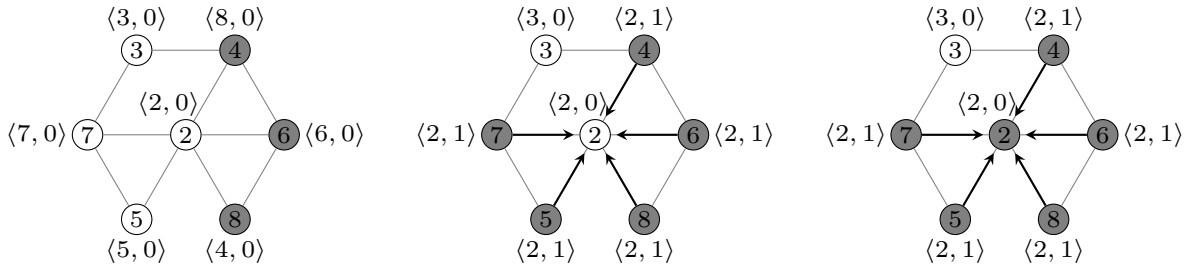
$Join(p, q) \equiv (IsFalseRoot(p) \vee (SuccKey(q) < p.key)) \wedge (q.key = BestNbrKey(p))$
 $\quad \wedge (FalseChldrn(p) = \emptyset) \wedge (q.color = 2)$
 $Reset(p) \equiv IsFalseRoot(p)$
 $Color1(p) \equiv (p.color = 2) \wedge (p.parent.color = 2) \wedge (\forall q \in TrueChldrn(p), q.color = 1)$
 $\quad \wedge (Recruits(p) = \emptyset) \wedge \neg ColorFrozen(p)$
 $Color2(p) \equiv (p.color = 1) \wedge (p.parent.color = 1) \wedge (\forall q \in TrueChldrn(p), q.color = 2)$
 $\quad \wedge \neg ColorFrozen(p)$
 $UpdateDone(p) \equiv p.done \neq Done(p)$

Actions

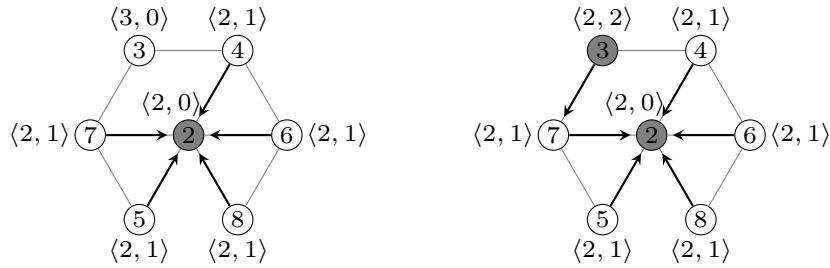
J (priority 1) $:: Join(p, q) \rightarrow p.key = SuccKey(q);$
 $\quad p.parent = q;$
 $\quad p.color = 1;$
 $\quad p.done = \mathbf{false};$
 R (priority 2) $:: Reset(p) \rightarrow p.key = SelfKey(p);$
 $\quad p.parent = p;$
 $\quad p.color = 2;$
 $\quad p.done = \mathbf{false};$
 $C1$ (priority 3) $:: Color1(p) \rightarrow p.color = 1;$
 $\quad p.done = Done(p);$
 $C2$ (priority 3) $:: Color2(p) \rightarrow p.color = 2;$
 $\quad p.done = Done(p);$
 UD (priority 4) $:: UpdateDone(p) \rightarrow p.done = Done(p);$



(a) Initial configuration. 1 is a fake ID. (b) 2 and 3 have executed J -action and chosen 7 (of color 2) as parent. 5 has changed its color to 2 executing $C2$ -action. (c) The tree of 7 was color locked. Then, 7 executed R -action.



(d) 2, 3 and 5 were false roots and have executed R -action. (e) 4, 5, 6, 7 and 8 have executed J -action and chosen 2 as parent. (f) 3 cannot join the tree of 2 because all its neighbors have color 1. 2 has changed its color to 1 executing $C1$ -action.



(g) 4,5,6,7 and 8 have changed their color to 2 executing $C2$ -action. (h) Then, 3 was able to execute J -action and join the tree of 2.

Figure 9: Example of execution of algorithm \mathcal{DLV} . The ID is represented inside the node. The label next to a node shows its key . The arrows represent $parent$ pointers. No arrow exit a node if its parent is itself. The filling represents the color: gray for 1 and white for 2.

- Layer 4: lines 1 to 7

We choose the following initial configuration. Notice that 0 is a fake ID smaller than every ID in the network.

- For $i \in [1 \dots 14]$, $j \in [1 \dots \beta]$, $p_{(i,j)}.leader = 0$, $p_{(i,j)}.level = i$ and $p_{(i,j)}.done = \mathbf{false}$
- For $j \in [1 \dots \beta]$,
 - $p_{(1,j)}.parent = p_{(1,j)}$
 - $p_{(8,j)}.parent = p_{(7,1)}$
 - $p_{(12,j)}.parent = p_{(11,1)}$
 - $p_{(14,j)}.parent = p_{(13,1)}$
 - For $i \in [2 \dots 7] \cup [9 \dots 11] \cup \{13\}$, $p_{(i,j)}.parent = p_{(i-1,j)}$
- For $i \in [1 \dots 14]$, $p_{(i,1)}.color = i \pmod{2} + 1$
- For $j \in [2 \dots \beta]$,
 - $p_{(8,j)}.color = p_{(12,j)}.color = p_{(14,j)} = 1$
 - For $i \in [1 \dots 7] \cup [9 \dots 11] \cup \{13\}$, $p_{(i,j)}.color = 2$

We consider an unfair daemon which selects the enabled processes according to function DAEMON (see Algo. 3). In this algorithm, $top(i)$ (respectively $bottom(i)$) is the number of the first line (respectively last line) of layer i . For example, $top(3) = 8$ and $bottom(3) = 11$.

The idea is to reset a branch of the tree and then, rebuild symmetrically the tree on the next column: a process chooses as parent the neighbor of smaller key, *i.e.*, the extreme left neighbor one line above having 0 as *leader*. More precisely, a first sequence of actions R resets the first column and the layer 1 (Fig. 11). Then, the layer 1 is rebuilt on the second column (BUILD(1,2)) and reset again (Fig. 12) *etc.* until the last column. Then, the tree is rebuilt since the second layer on the second column (BUILD(2,2)) and the extreme left branch is reset (Fig 13) and so on.

We count how many times processes $p_{(14,.)}$ executes R action:

- Each process $p_{(14,.)}$ executes once action R on line 12 of RESET(*layer*, *column*), when *layer* = 1.
- RESET(3, *column*) is called β times by DAEMON.
- RESET(2, *column*) is called β times by RESET(3, *column*).
- RESET(1, *column*) is called β times by RESET(2, *column*).

Hence, R action is executed β^4 times by the processes of line 14. Now, $\beta = n/14$. Hence, the computation lasts $\Omega(n^4)$ steps.

5.3 Generalization to an example in $\Omega(n^{\alpha+1})$ steps

Starting from E_α ($\alpha \geq 4$), an example in $\Omega(n^\alpha)$ steps, we can build $E_{\alpha+1}$, an example in $\Omega(n^{\alpha+1})$ steps, based on the same principle as in Subsection 5.2, by adding a layer. If E_α has $L\beta$ processes $p_{(i,j)}$ ($1 \leq i \leq L$, $1 \leq j \leq \beta$), then $E_{\alpha+1}$ has $L' = 2L$ lines of β processes $q_{(i',j')}$ ($1 \leq i' \leq L'$, $1 \leq j' \leq \beta$). The construction principle is as follows:

- We increases the *level* and the ID of the $L\beta$ processes of E_α as follows: $\forall i \in [1 \dots L]$, $\forall j \in [1 \dots \beta]$, $q_{(i+L,j)} = p_{(i,j)}$. The ID of $q_{(i+L,j)}$ becomes $(i+L-1)\beta + j$ and $q_{(i+L,j)}.level = i+L$. The value of variables *parent*, *color* and *done* do not change.
- At the top of E_α , we add L lines of β processes. These new processes satisfy:
 - $\forall i \in [1 \dots L]$, $\forall j \in [1 \dots \beta]$, $q_{(i,j)}.id = (i-1)\beta + j$, $q_{(i,j)}.leader = 0$, $q_{(i,j)}.level = i$ and $q_{(i,j)}.done = \mathbf{false}$.
 - $\forall i \in [2 \dots L]$, $\forall j \in [1 \dots \beta]$, $\{q_{(i,j)}, q_{(i-1,j)}\} \in V$ and $q_{(i,j)}.parent = q_{(i-1,j)}$.

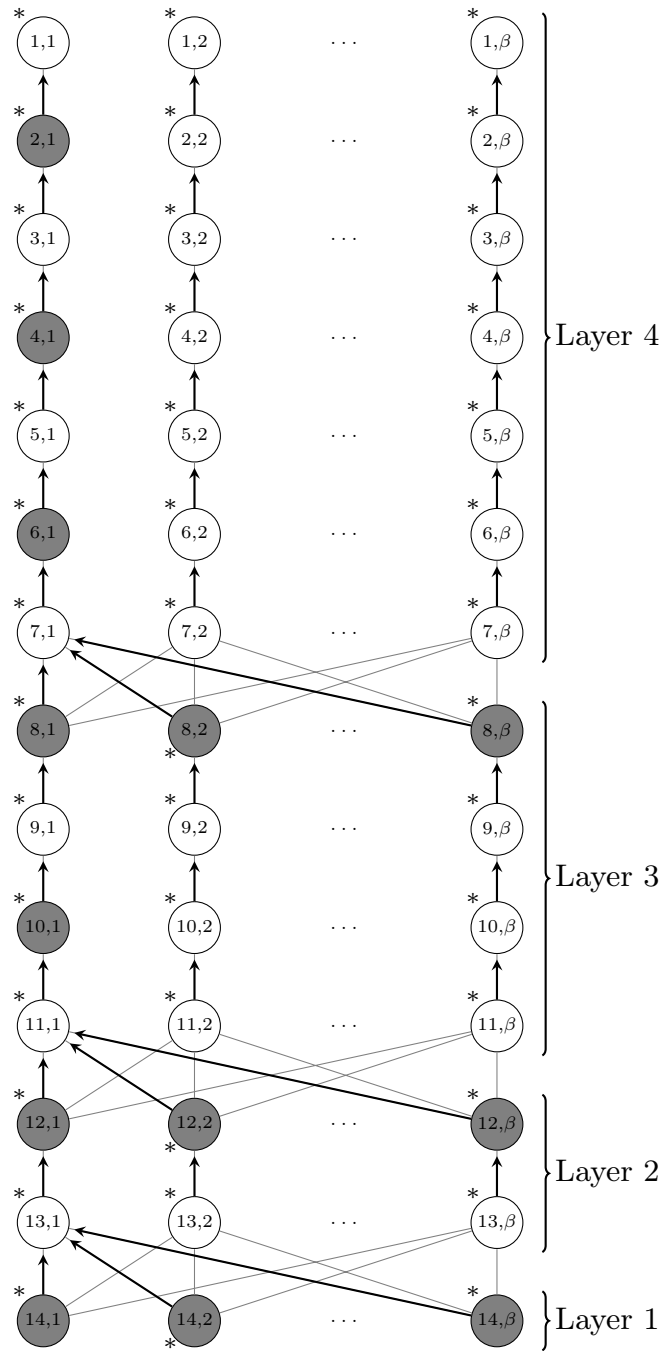


Figure 10: Initial configuration. The *leader* of a process is 0 if it gets a star or its own ID otherwise. *level* is not represented as it is always correct.

Algorithm 3 Algorithm of the daemon.

```
1: function DAEMON
2:   for  $i = 1 \dots \beta$  do
3:     RESET(3,i);
4:     if  $i < \beta$  then
5:       BUILD(3,i+1);
6:
7: function RESET(layer, column)
8:   for  $i = \text{top}(\text{layer} + 1) \dots \text{bottom}(\text{layer} + 1)$  do
9:      $p_{(i,\text{column})}$  executes  $R$ ;
10:  if layer = 1 then
11:    for  $j = 1 \dots \beta$  do
12:       $p_{(L,j)}$  executes  $R$ ;
13:  else
14:    for  $j = 1 \dots \beta$  do
15:      RESET(layer - 1,  $j$ );
16:      if  $j < \beta$  then
17:        BUILD(layer - 1,  $j + 1$ );
18:
19: function BUILD(layer, column)
20:   for  $i = \text{top}(\text{layer}) \dots \text{bottom}(\text{layer})$  do
21:     for  $j = 1 \dots \beta$  do
22:        $p_{(i,j)}$  executes  $J$ ;
23:     for  $k = i - 1 \dots 2(i - \frac{L}{2})$  do
24:       if  $k \geq \text{top}(\text{layer})$  then
25:         for  $j = 1 \dots \beta$  do
26:            $p_{(k,j)}$  executes  $C1$ ;
27:       else
28:          $p_{(k,\text{column})}$  executes  $C1$ ;
29:     for  $k = i \dots 2(i - \frac{L}{2}) + 1$  do
30:       if  $k \geq \text{top}(\text{layer})$  then
31:         for  $j = 1 \dots \beta$  do
32:            $p_{(k,j)}$  executes  $C2$ ;
33:       else
34:          $p_{(k,\text{column})}$  executes  $C2$ ;
35:   if layer > 1 then
36:     BUILD(layer - 1, 1);
```

▷ Reset of layer 1, $L = \text{top}(1) = 14$

▷ step -1

▷ step -1

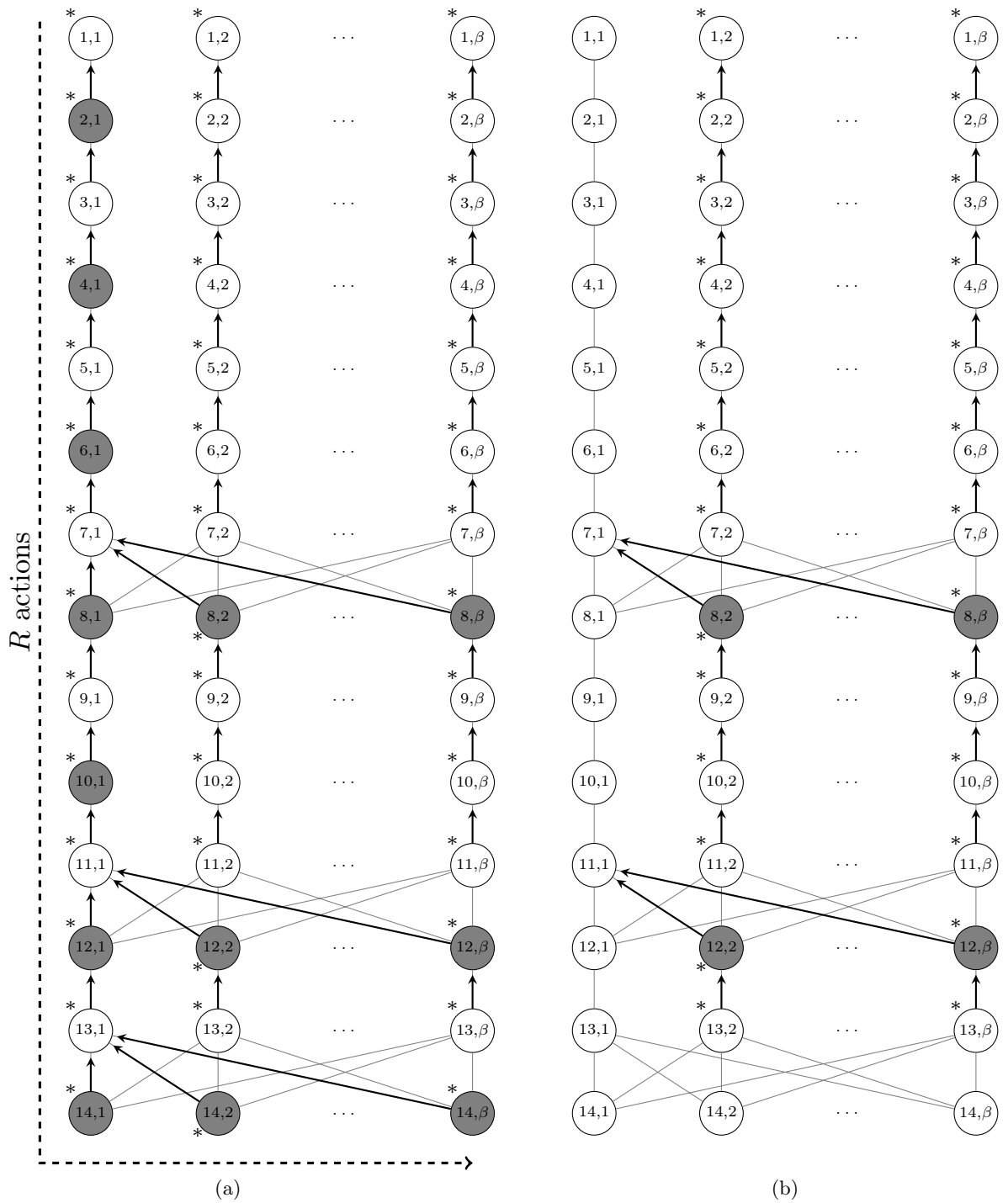


Figure 11: First sequence of R actions that resets the left column and the layer 1.

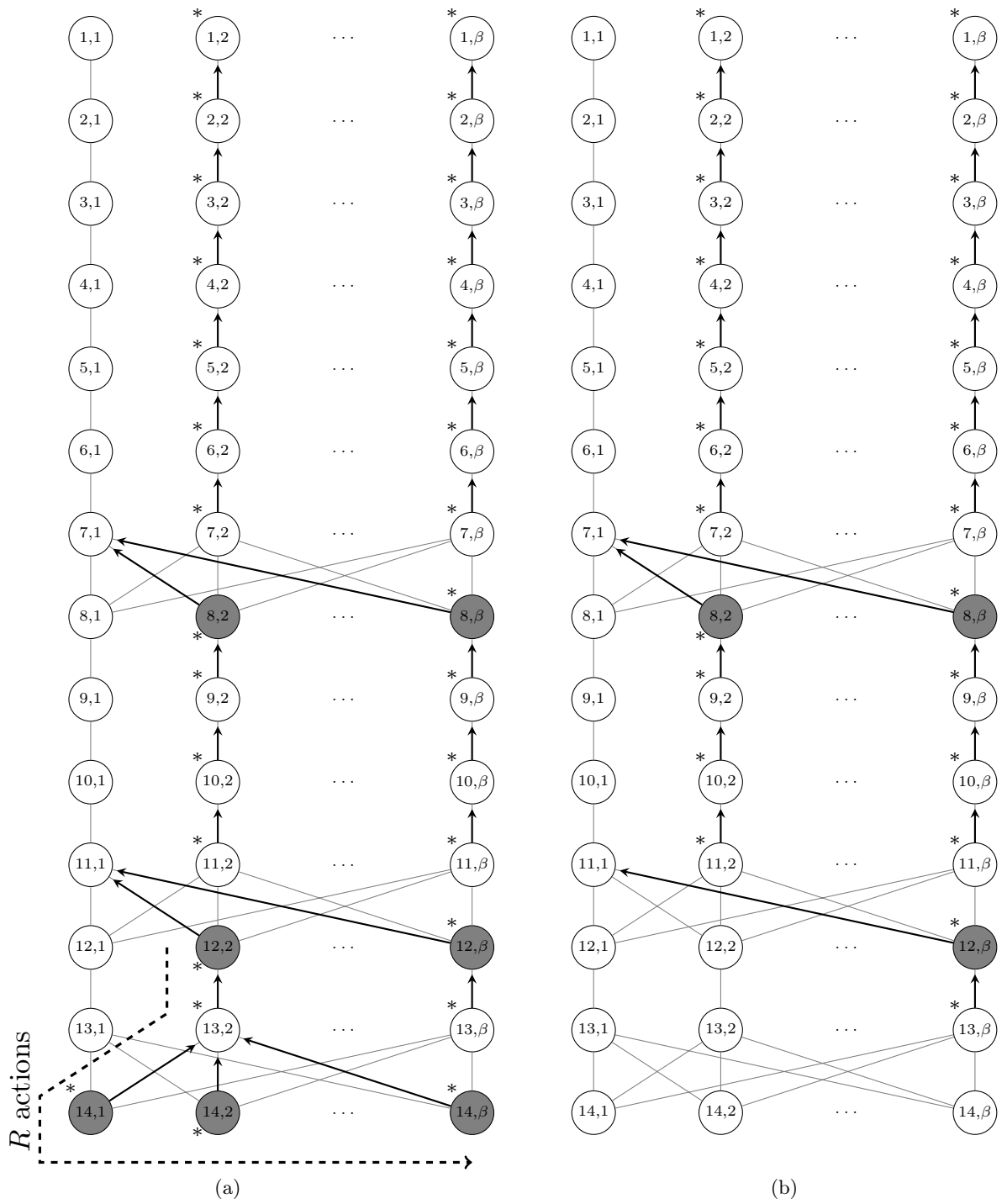


Figure 12: Reconstruction of the layer 1 on the second column and reset.

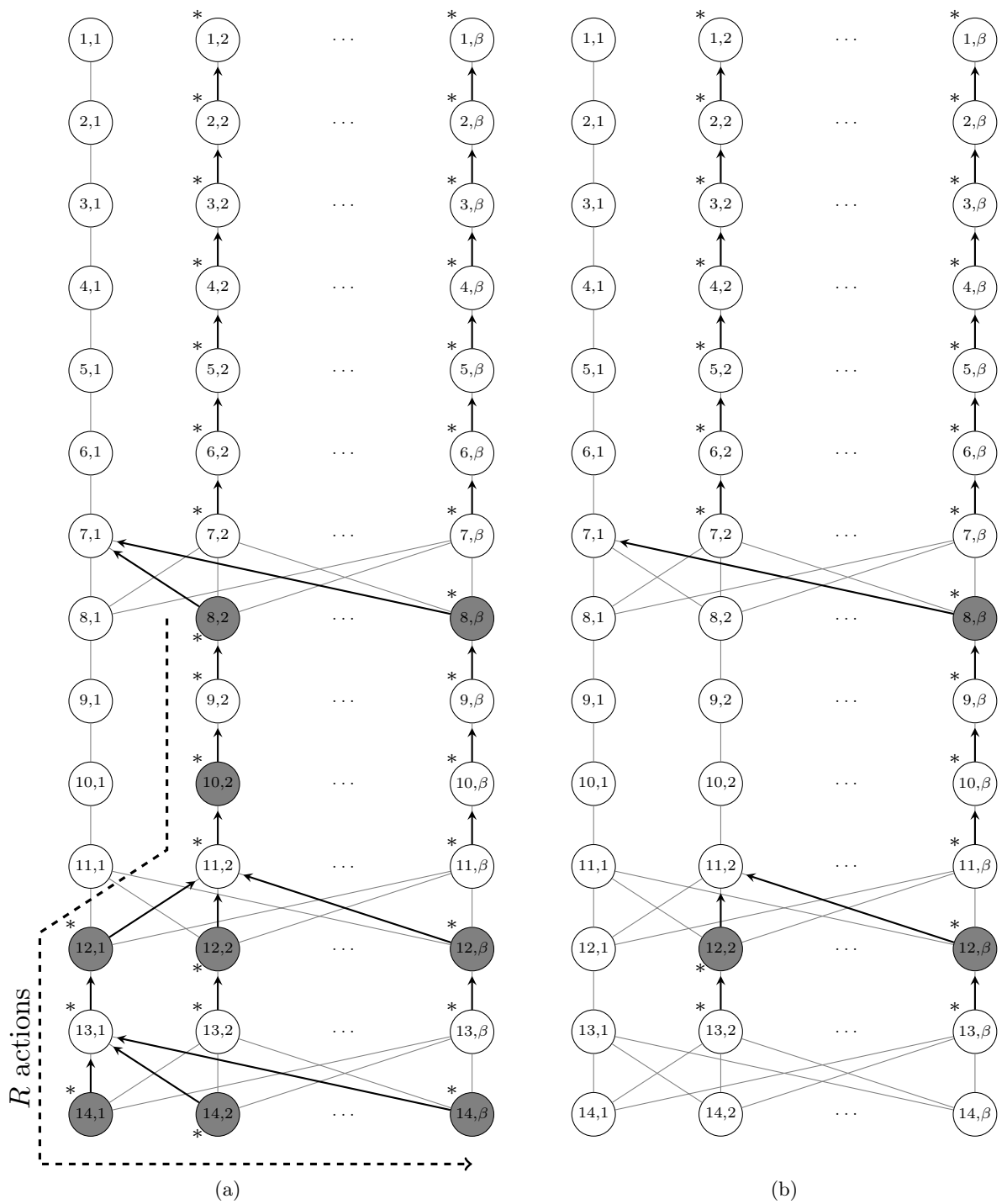


Figure 13: Reconstruction of the layer 2 on the second column and reset.

- $\forall j = [1 \dots \beta], q_{(1,j)}.parent = q_{(1,j)}$.
- $\forall j = [2 \dots \beta], \forall i \in [1 \dots L], q_{(i,j)}.color = 2$.
- $\forall i \in [1 \dots L], q_{(i,1)}.color = i \pmod{2} + 1$.

• The former first line of E_α becomes a new layer with the last added line:

- $\forall j \in [1 \dots \beta], \forall j' \in [1 \dots \beta], \{q_{(L,j)}, q_{(L+1,j')}\} \in V$.
- $\forall j \in [1 \dots \beta], q_{(L+1,j)}.parent = q_{(L,1)}$.

Fig. 14 shows the structure of the network for E_5 and its initial configuration.

Then, the daemon selects processes according to function $DAEMON(\alpha + 1)$ (see Algo. 4) which is the generalization of the algorithm presented in section 5.2. In E_α , processes $p_{(L,\cdot)}$ execute β^α times R action. Now, we added a new level of recursion. Then, processes $q_{(L',\cdot)}$ execute $\beta^{\alpha+1}$ times R action. $\beta = \frac{n}{L'}$ hence the computation lasts $\Omega(n^{\alpha+1})$ steps.

Algorithm 4 Generalization of the algorithm of the daemon for $E_{\alpha+1}$.

```

1: function DAEMON( $\alpha + 1$ )
2:   for  $i = 1 \dots \beta$  do
3:     RESET( $\alpha, i$ ); ▷ See Algo. 3
4:     if  $i < \beta$  then
5:       BUILD( $\alpha, i+1$ ); ▷ See Algo. 3

```

Then, for every $\alpha \geq 3$, we can build a network $E_{\alpha+1}$ such that there is a computation that lasts $\Omega(n^{\alpha+1})$ steps. So, the stabilization time of \mathcal{DLV} in steps is not polynomial.

6 Conclusion

We proposed a silent self-stabilizing leader election algorithm, called \mathcal{LE} , for bidirectional connected identified networks of arbitrary topology. Starting from any arbitrary configuration, \mathcal{LE} converges to a terminal configuration, where all processes know the ID of the leader, this latter being the process of minimum ID. Moreover, as in most of the solutions from the literature, a distributed spanning tree rooted at the leader is defined in the terminal configuration.

\mathcal{LE} is written in the locally shared memory model. It assumes the distributed unfair daemon, the most general scheduling hypothesis of the model. Moreover, it requires no global knowledge on the network (such as an upper bound on the diameter or the number of processes, for example). \mathcal{LE} is asymptotically optimal in space, as it requires $\Theta(\log n)$ bits per process, where n is the size of the network. We analyzed its stabilization time both in rounds and steps. We showed that \mathcal{LE} stabilizes in at most $3n + \mathcal{D}$ rounds, where \mathcal{D} is the diameter of the network. We also proved that for every $n \geq 4$, there is a network of n processes, from which a possible execution exactly lasts this complexity.

Finally, we proved that \mathcal{LE} achieves a stabilization time polynomial in steps. More precisely, its stabilization time is at most $\frac{n^3}{2} + 2n^2 + \frac{n}{2} + 1$ steps. Then, we showed for every $n \geq 4$, there exists a network of n processes, from which a possible execution exactly lasts $\frac{n^3}{6} + \frac{5}{2}n^2 - \frac{11}{3}n + 2$ steps, establishing then that the worst case is in $\Theta(n^3)$.

Perspectives of this work deal with complexity issues. In [8], Datta *et al* showed that it is easy to implement a silent self-stabilizing leader election which works assuming an unfair daemon, uses $\Theta(\log n)$ bits per process, and stabilizes in $O(D)$ rounds (where D is an upper bound on \mathcal{D}), yet *if processes have knowledge of D* . Now, it is worth investigating if it is possible to design an algorithm which works assuming an unfair daemon, uses $\Theta(\log n)$ bits per process, and stabilizes in $O(\mathcal{D})$ rounds without using any global knowledge. We believe this problem remains difficult, even adding some fairness assumption.

References

- [1] Yehuda Afek and Anat Bremler-Barr. Self-Stabilizing Unidirectional Network Algorithms by Power Supply. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

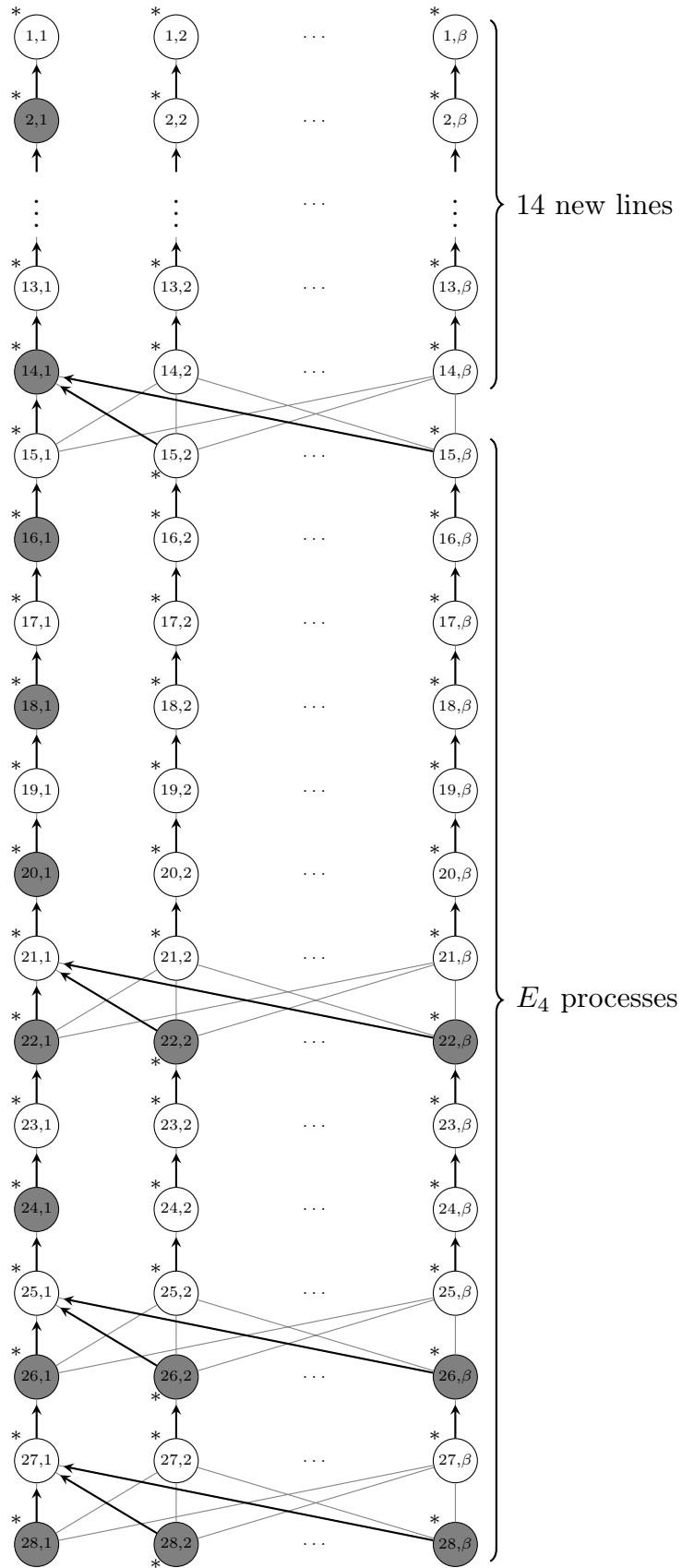


Figure 14: Initial configuration of the example in $O(n^5)$ steps.

- [2] Anish Arora and Mohamed G. Gouda. Distributed Reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.
- [3] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time Optimal Self-stabilizing Synchronization. In *STOC*, pages 652–661, 1993.
- [4] Janna Burman and Shay Kutten. Time Optimal Asynchronous Self-stabilizing Spanning Tree. In *DISC*, pages 92–107, 2007.
- [5] Ernest J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.
- [6] Ajoy Kumar Datta, Lawrence L. Larmore, and Hema Piniganti. Self-stabilizing Leader Election in Dynamic Networks. In *SSS*, pages 35–49, 2010.
- [7] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. An $O(n)$ -time Self-stabilizing Leader Election Algorithm. *J. Parallel Distrib. Comput.*, 71(11):1532–1544, 2011.
- [8] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing Leader Election in Optimal Space under an Arbitrary Scheduler. *Theor. Comput. Sci.*, 412(40):5541–5561, 2011.
- [9] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.
- [10] Shlomi Dolev. *Self-stabilization*. MIT Press, March 2000.
- [11] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory Requirements for Silent Stabilization. *Acta Inf.*, 36(6):447–462, 1999.
- [12] Shlomi Dolev and Ted Herman. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.
- [13] Alex Kravchik and Shay Kutten. Time Optimal Synchronous Self Stabilizing Spanning Tree. In *DISC*, pages 91–105, 2013.
- [14] Adrian Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.