Self-Stabilizing Leader Election in Polynomial Steps*

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

[§]VERIMAG UMR 5104, Université Grenoble Alpes, France
[†]MIS Lab., Université Picardie Jules Verne, France
[‡]LIP6 UMR 7606, INRIA, UPMC Sorbonne Universitès, France

Abstract

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. 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 [10, 9] designed with similar settings. To the best of our knowledge, this is the first self-stabilizing leader election algorithm for arbitrary identified networks that is proven to achieve a stabilization time polynomial in steps. By contrast, we show that the previous best existing algorithms designed with similar settings [10, 9] stabilize in a non polynomial number of steps in the worst case.

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

1 Introduction

In distributed computing, the *leader election* problem consists in distinguishing a single 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 [11, 12] 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

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

one. Notice that efficiency of self-stabilizing algorithms is mainly evaluated according to their stabilization time and memory requirement.

We consider deterministic[1] 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 [11]. In this model, the distributed unfair daemon is known as the weakest scheduling assumption. Under such an assumption, proving that a given algorithm is self-stabilizing implies that the stabilization time must be finite in terms of atomic steps. However, despite some solutions assuming all these settings (in particular the unfairness assumption) are available in the literature [10, 9, 8], none of them is proven to achieve a polynomial upper bound in steps on its stabilization time. Actually, the time complexities of all these solutions are analyzed in terms of rounds only.

Related Work In [13], Dolev *et al* showed that silent self-stabilizing leader election requires $\Omega(\log n)$ bits per process, where n is the number of processes. Notice that *non-silent* self-stabilizing leader election can be achieved using less memory, *e.g.*, the non-silent self-stabilizing leader election algorithm for unoriented ring-shaped networks given in [5] requires $O(\log \log n)$ space per process.

Self-stabilizing leader election algorithms for arbitrary connected identified networks have been proposed in the message-passing model [1, 3, 6]. First, the algorithm of Afek and Bremler [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, 6]. However, both solutions assume that processes know some upper bound D on the diameter $\mathcal{D}$; and require $\Theta(\log D \log n)$ bits per process.

Several solutions are also given in the shared memory model [14, 2, 8, 10, 9, 17]. The algorithm proposed by Dolev and Herman [14] 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* [8] propose the first self-stabilizing leader election algorithm (for arbitrary connected identified networks) proven under the distributed unfair daemon. This algorithm stabilizes in $O(\mathcal{D})$ 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 [10, 9, 17] have a memory requirement which is asymptotically optimal (*i.e.* in $\Theta(\log n)$). The algorithm proposed by Kravchik and Kutten [17] 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 [10, 9] assume a distributed unfair daemon and have a stabilization time in $O(n)$ rounds. However, despite these two algorithms stabilizing within a finite number of steps (indeed, they are proven assuming an unfair daemon), no step complexity analysis is proposed.

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 the network.

[1]We only consider here deterministic algorithms.

Like previous solutions of the literature [10, 9], it is asymptotically optimal in space (*i.e.*, it can be implemented 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 algorithms given in [10, 9], noted here $\mathcal{DLV}$ and $\mathcal{DLV}2$, respectively. These latter are the closest to ours in terms of performance. We show that their stabilization time is not polynomial. Indeed, there is no constant α such that the stabilization time of $\mathcal{DLV}$ is in $O(n^\alpha)$ steps. More precisely, we show that fixing α to any constant greater than or equal to 4, for every $\beta \geq 2$, there exists a network of $n = 2^{\alpha-1} \times \beta$ processes in which there exists a possible execution that stabilizes in $\Omega(n^\alpha)$ steps. Similarly, for $n \geq 5$, there exists a network and a possible execution of $\mathcal{DLV}2$ that stabilizes in $\Omega(2^{\lfloor \frac{n-1}{4} \rfloor})$ steps.

Roadmap The next section is dedicated to 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 algorithms given in [10, 9] are not polynomial in steps in Sections 5 and 6, respectively. We present some experimental results in Section 7. We conclude in Section 8.

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, 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. (So, the minimum ID will be also denoted by ℓ.)

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 only write to its own variables. The *state* of a process is the vector of values of all its variables. A configuration γ of the system is a vector consisting in one state of each process. We denote by $\gamma(p)$ the state of 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:

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

The *labels* are used to identify actions. The *guard* of an action in the program of 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. By extension, 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 if 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 there is at least one enabled process, 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 $\gamma_0, \gamma_1, \ldots$ 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.

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* [15]. This latter allows to highlight the execution time according to the speed of the slowest process. If a process p 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 $\gamma_i \mapsto \gamma_{i+1}$. Neutralization of p is caused by the following situation: at least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change makes the guards of all actions of p false. The first round of an execution e, noted e', is the minimal prefix of e in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let e'' be the suffix of e starting from the last configuration of e'. The second round of e is the first round of e'', and so forth.

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

Every configuration that is not legitimate is called *illegitimate*. The *stabilization time* is the maximum time (in steps or rounds) to reach a legitimate configuration starting from any configuration.

2.5 Self-stabilizing Leader Election

We define SP_{LE} 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. An execution $e \in \mathcal{E}$ satisfies SP_{LE} if and only if:

1. For each 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 each 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}.

An algorithm $\mathcal{A}$ is *silent* if and only if every execution is finite [13]. 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: (1) in every terminal configuration γ, $\forall p, q \in V$, $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process; (2) 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* (*idR* means ID of the root). Formally:

1. $\ell.idR = \ell$, $\ell.par = \ell$, and $\ell.level = 0$, and

2. $\forall p \neq \ell$, $p.idR = \ell$, $p.par$ points to the parent of p in the tree and $p.level$ is the level of p in the tree.

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, it 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:

$$SelfRoot(p) \equiv (p.par = p)$$

$$SelfRootOk'(p) \equiv (p.level = 0) \wedge (p.idR = p)$$

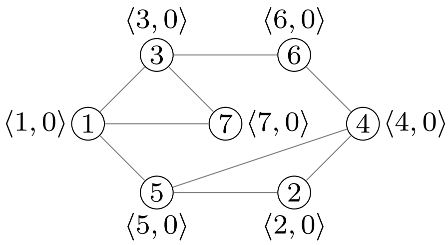
Note that, in the sequel, we say that p is a *self root* when $SelfRoot(p)$ holds.

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

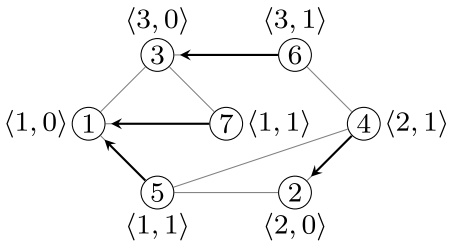
$$\begin{array}{lllll} J\text{-}Action' & :: & \exists q \in \mathcal{N}_p, (q.idR < p.idR) & \to & p.par \leftarrow \min_{\preceq}\{q \in \mathcal{N}_p\}; \\ & & & & p.idR \leftarrow p.par.idR; \\ & & & & p.level \leftarrow p.par.level + 1; \end{array}$$

where $\forall x, y \in V, x \preceq y \Leftrightarrow (x.idR \leq y.idR) \wedge [(x.idR = y.idR) \Rightarrow (x < y)]$.

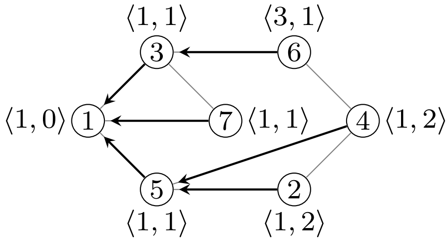
Informally, when p discovers that $p.idR$ is not equal to the minimum identifier, it updates its variables accordingly: let q be the neighbor of p having idR minimum. Then, p selects q as new parent (*i.e.*, $p.par \leftarrow q$ and $p.level \leftarrow p.par.level + 1$) and sets $p.idR$ to the value of $q.idR$.



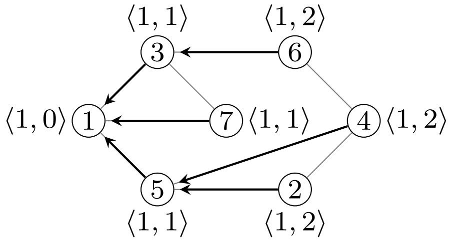
(a) Initial configuration. $SelfRoot(p) \wedge SelfRootOk'(p)$ holds for every process p.



(b) 4, 5, 6, and 7 have executed *J-Action*$'$. Note that *J-Action*$'$ was not enabled at 2 because it is a local minimum.



(c) 2, 3, and 4 have executed *J-Action*$'$. 3 joins the tree rooted at 1. However, the new value of $3.idR$ is not yet propagated to its child 6.



(d) 6 has executed *J-Action*$'$. The configuration is now terminal, $\ell = 1$ is elected, and a tree rooted at ℓ is available.

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.

If there are several neighbors having idR minimum, we break ties using the identifiers of those neighbors.

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.

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

(a) Illegitimate initial configuration, where 2 and 5 have fake idR.

(b) 3 and 4 executed $J\text{-}Action'$. The configuration is now terminal.

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

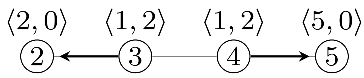
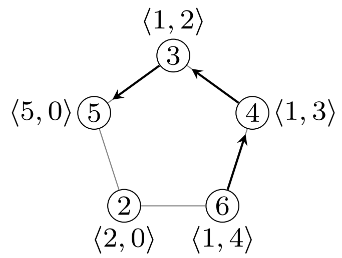


Figure 3: One step after Figure 2b, 2 and 5 have reset.

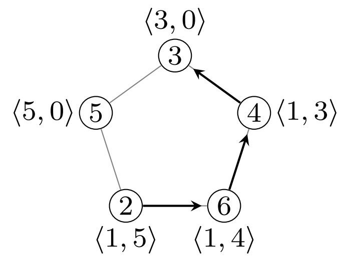
The existence of fake IDs may lead the system to an illegitimate terminal configuration. Refer to the example of execution given in Figure 2: starting from the configuration in 2a, if processes 3 and 4 move, the system reaches the terminal configuration given in 2b, where there are two trees and the idR variables elect the fake ID 1. In this example, 2 and 5 can detect the problem. Indeed, predicate $SelfRootOk'$ is violated by both 2 and 5. One may believe that it is sufficient to reset the local state of processes which detect inconsistency (here processes 2 and 5) to $p.idR \leftarrow p$, $p.par \leftarrow p$ and $p.level \leftarrow 0$. After these resets, there are still some errors, as shown on Figure 3. Again, 3 and 4 can detect the problem. Indeed, predicate $GoodIdR(p, p.par) \wedge GoodLevel(p, p.par)$ is violated by both 3 and 4. In this example, after 3 and 4 have reset, all inconsistencies have been removed. So let define the following action:

$$
\begin{aligned}
\textit{R-Action}' \quad :: \quad & \big(\mathit{SelfRoot}(p) \wedge \neg \mathit{SelfRootOk}'(p)\big) \vee \big(\neg \mathit{SelfRoot}(p) \\
& \wedge \neg(\mathit{GoodIdR}(p, p.par) \wedge \mathit{GoodLevel}(p, p.par))\big) \\
& \qquad \to p.par \leftarrow p; p.idR \leftarrow p; p.level \leftarrow 0;
\end{aligned}
$$

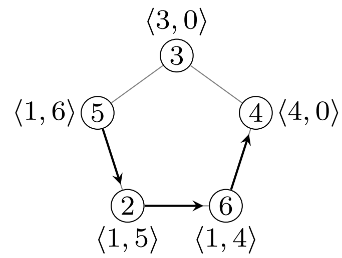
Unfortunately, this additional action does not ensure the convergence in all cases, see the example in Figure 4. Indeed, if a process resets, it becomes a self root but this does not erase the fake ID in the rest of its subtree. Then, another process can join the tree and adopt the fake ID which will be 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 freeze 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.



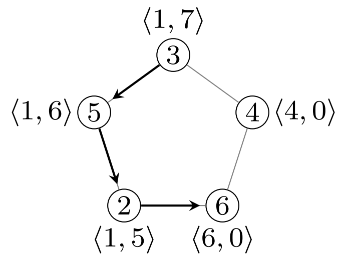
(a) Illegitimate initial configuration.



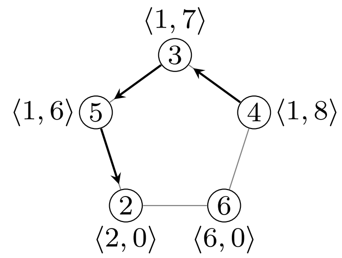
(b) 2 joins the tree. 3 leaves it.



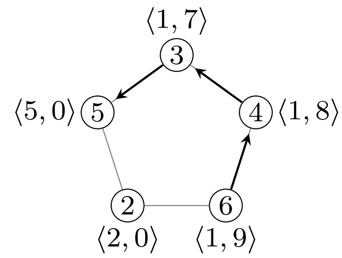
(c) 5 joins the tree. 4 leaves it.



(d) Both 3 and 6 move.



(e) 4 joins, 2 leaves.



(f) Configuration similar to 4a

Figure 4: The first process of the chain of bold arrows violates the predicate $\mathit{SelfRootOk}'$ and resets by executing *R-Action*$'$, while another process joins its tree. This cycle of resets and joins might never terminate.

3.3 Cleaning Abnormal Trees

To introduce the trees, we define what is a "good relation" between a parent and its children. Namely, the predicate $\mathit{KinshipOk}'(p, q)$ models that a process p is a *real child* of its parent $q = p.par$. This predicate holds if and only if $\mathit{GoodLevel}(p, q)$ and $\mathit{GoodIdR}(p, q)$ are true. This relation defines a spanning forest: a *tree* is a maximal set of processes connected by *par* pointers and satisfying $\mathit{KinshipOk}'$ relation. A process p is a root of such a tree whenever $\mathit{SelfRoot}(p)$ holds or $\mathit{KinshipOk}'(p, p.par)$ is false. When $\mathit{SelfRoot}(p) \wedge \mathit{SelfRootOk}'(p)$ is true, p is a *normal root* just as in the non self-stabilizing case (see 3.1). In other cases, there is an error and p is said to be an *abnormal root*:

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

These are the two possible errors identified in the Subsection 3.2. A tree is called an *abnormal tree* (resp. *normal*) when its root is abnormal (resp. normal).

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

Algorithm 1 Algorithm $\mathcal{LE}$ for every 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

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

Predicates

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

Guards

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

Actions

$$
\begin{array}{lllll}
EB\text{-}action & :: & EBroadcast(p) & \rightarrow & p.status \leftarrow EB; \\
EF\text{-}action & :: & EFeedback(p) & \rightarrow & p.status \leftarrow EF; \\
R\text{-}action & :: & Reset(p) & \rightarrow & p.status \leftarrow C; p.par \leftarrow p; \\
 & & & & p.idR \leftarrow p; p.level \leftarrow 0; \\
J\text{-}action & :: & Join(p) \wedge \neg EBroadcast(p) & \rightarrow & p.par \leftarrow Min_p; p.idR \leftarrow p.par.idR; \\
 & & & & p.level \leftarrow p.par.level + 1;
\end{array}
$$

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 (inspired from [4]) is achieved using an additional variable, *status*, that is used as follows. If a process is clean (*i.e.*, not involved into any freezing operation), then its *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" [7, 18] in 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 said 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 the one of q. This is expressed with the predicate $GoodStatus(p, q)$ which is used to enforce the $KinshipOk(p, q)$ relation:

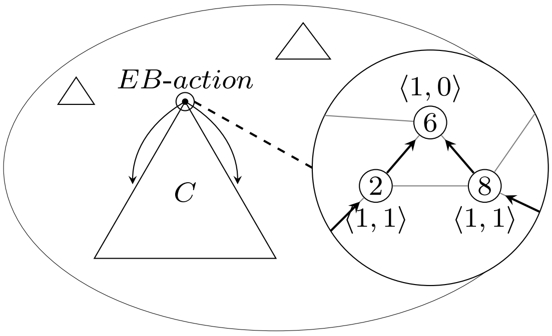
$$\begin{aligned} GoodStatus(p,q) &\equiv [(p.status = EB) \Rightarrow (q.status = EB)] \\ &\quad \wedge[(p.status = EF) \Rightarrow (q.status \neq C)] \\ &\quad \wedge[(p.status = C) \Rightarrow (q.status \neq EF)] \\ KinshipOk(p,q) &\equiv KinshipOk'(p,q) \wedge GoodStatus(p,q) \end{aligned}$$

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.

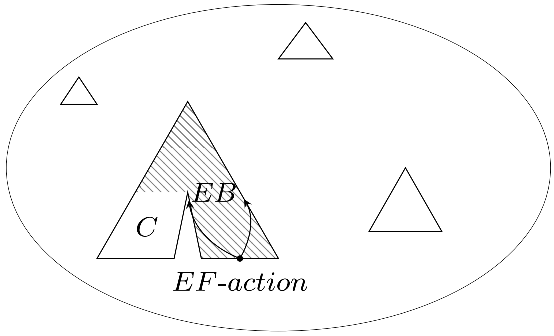
Normal execution Remark that, after all abnormal trees have been removed, all processes have status C and the algorithm works as in the initial version. Notice that the guard of *J-action* has been enforced so that only processes with status C and which are not abnormal root can execute it, and when 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 5 shows how an abnormal tree is cleaned. In the first phase (see Figure 5a), 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 5b). 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 5c). 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 5e). Its former real children (of status EF) become themselves abnormal roots of dead trees (Figure 5f) and reset, *etc.*

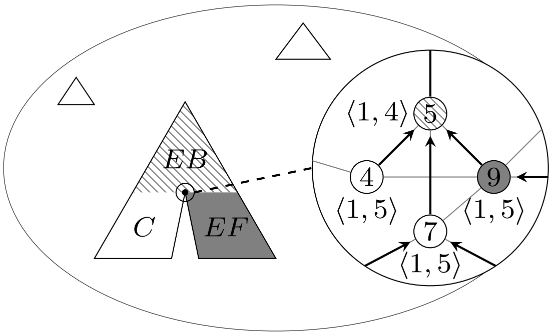
Finally, we used the predicate *Allowed*(p) to temporarily lock the parent of p in two particular situations — illustrated in Figure 6 — where p is enabled to switch its status from C to EB. These locks impact neither the correctness nor the complexity of $\mathcal{LE}$. Rather, they allow us to simplify the proofs by ensuring that, once enabled, *EB-action* remains continuously enabled until executed.



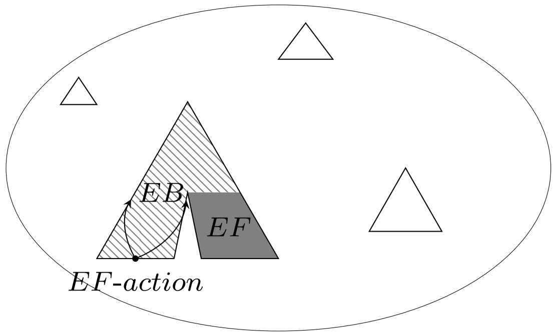
(a) When an abnormal root detects an error, it executes EB-*action*. The EB-wave is broadcast to the leaves. Here, 6 is an abnormal root because it is a self root and its idR is different from its ID ($1 \neq 6$).



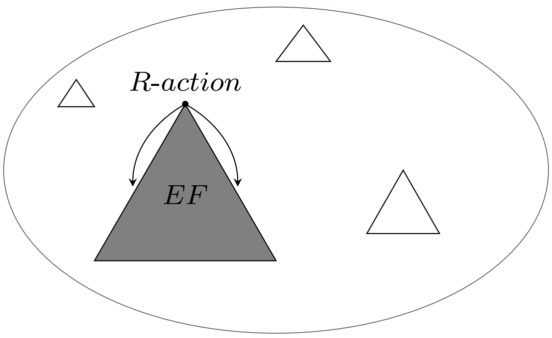
(b) When the EB-wave reaches a leaf, it executes EF-*action*. The EF-wave is propagated up to the root.



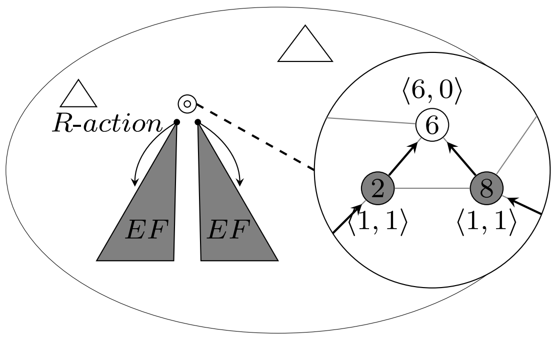
(c) It may happen that the EF-wave reaches a node, here process 5, even though the EB-wave is still broadcasting into some of its proper subtrees: 5 must wait that the status of 4 and 7 become EF before executing EF-*action*.



(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 5: 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 6: Example of situations where the parent of a process is locked.

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 $p_k, p_{k-1}, \ldots, p_0$ such that $p_{i-1} \in \mathcal{N}_{p_i}$, for all i in $\{1, ..., k\}$. 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 $p_k, p_{k-1}, \ldots, p_0$ 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 of 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 (Graph of Kinship Relations). *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 $p_k, p_{k-1}, \ldots, p_0$ such that for all i in $\{1, ..., k\}$, $(p_i, p_{i-1}) \in KR$.

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 if $SelfRoot(p) \vee \neg KinshipOk(p, p.par)$ holds in γ.*

Next, we define the paths, called $KPath$s, 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 $p_0, p_1, \ldots, p_k$ 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:*

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

This means, in particular, that we identify each tree with the ID of its root.

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 $p_0, p_1, \ldots, p_k$, we define:*

$$S\text{-}Trace(p_0, p_1, \ldots, p_k) \in \{C, EB, EF\}^*$$

as the sequence $(p_0.status).(p_1.status) \ldots (p_k.status)$ 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. 7. Then, we prove that each segment contains a finite number of *J-action*s (Lemma 10). This latter result implies that every execution contains a finite number of *J-action*s (Corollary 2). Then, we show, in Lemma 11 and Corollary 3, 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 every configuration γ and every process p, any tree rooted at p such that $\neg AbRoot(p)$ in γ is called a* normal tree*. In this case, $SelfRoot(p) \wedge SelfRootOk(p)$ holds in γ, by Definition 3. Any tree that is not normal is simply said to be* abnormal*.*

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, p is said to be* dead*. A tree T in γ is called an* alive tree *in γ if and only if $\exists p \in T$ 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, but in a different tree T' in γ' (namely, the roots of T and T' are different), 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, p is a normal root ($SelfRoot(p) \wedge SelfRootOk(p)$ holds in γ), or p is not a root ($KinshipOk(p, p.par)$ holds in γ).

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

If p executes *R-action*, the predicate $SelfRoot(p) \wedge SelfRootOk(p)$ holds in γ'. So, p is a normal root in γ'.

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

- If q does not move in $\gamma \mapsto \gamma'$, then $\gamma'(p).par = q$, $\gamma'(q).status = C = \gamma'(p).status$, $\gamma'(p).level = \gamma(q).level + 1 = \gamma'(q).level + 1$, and $\gamma'(p).idR = \gamma(q).idR = \gamma'(q).idR < \gamma(p).idR \leq p$. Hence, the predicate $KinshipOk(p, p.par)$ is true in γ'. Now, we already know that $\neg SelfRoot(p)$ holds in γ'. Thus, $\neg SelfRoot(p) \wedge KinshipOk(p, q)$ holds in γ': p is not a root in γ', by Definition 3.

- Assume now that q moves during the step $\gamma \mapsto \gamma'$. 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, $\gamma'(p).idR = \gamma(q).idR \geq \gamma'(q).idR$ and $\gamma'(p).idR = \gamma(q).idR < \gamma(p).idR \leq p$. So, the predicate $GoodIdR(p, q)$ holds in γ'.

  If q executes *J-action*, then $\gamma'(p).idR \neq \gamma'(q).idR$. Otherwise, q executes *EB-action*, so $\gamma'(p).idR = \gamma'(q).idR$ and $\gamma'(p).level = \gamma(q).level + 1 = \gamma'(q).level + 1$. Hence, $GoodLevel(p, q)$ holds in γ'.

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

  Thus, $\neg SelfRoot(p) \wedge KinshipOk(p, q)$ holds in γ' and, so, p is not a root in γ', by Definition 3.

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 $GoodStatus(p, p.par)$ holds in γ, this implies that the status of $\gamma(p).par$ is either EB or C. Looking at case EB, p is a real child of $p.par$ in γ with status C; hence *EF-action* is disabled for $p.par$ in γ. Looking at case C, $p.par$ can execute *EB-action* and can change only its status to EB in $\gamma \mapsto \gamma'$: $GoodStatus(p, p.par)$ holds in γ' and consequently $KinshipOk(p, p.par)$ holds in γ'. $p.par$ can also execute *J-action* in $\gamma \mapsto \gamma'$. This means that in γ and in γ', $p.par$ has status C, hence $GoodStatus(p, p.par)$ holds in γ'. Furthermore, $p.par$ has a smaller value of idR in γ', so $GoodIdR(p, p.par)$ and $GoodLevel(p, p.par)$ are satisfied in γ', and consequently $KinshipOk(p, p.par)$ holds in γ'. □

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

Proof. Let $\gamma \mapsto \gamma'$ be 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 $AbRoot(p)$ holds in γ, 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 $\neg AbRoot(p)$ holds in γ, then p does not satisfy $SelfRoot(p)$. Indeed, the predicate $SelfRootOk(p)$ implies that $p.status = C$ in γ, a contradiction.
So, let $q = \gamma(p).par \in \mathcal{N}_p$. $\neg AbRoot(p)$ in γ implies that $q.status = EB$ and the predicate $KinshipOk(p, q)$ holds 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, the predicate $\neg AbRoot(p)$ still holds in γ', a contradiction.

$\gamma(p).status = C$**:** $\neg AbRoot(p)$ holds in γ (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. Let $q = \gamma(p).par$. Now, $p.par \neq p$ and $\neg AbRoot(p)$ 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-action*s To show that every process p executes only a finite number of *J-action*s, we prove below that p can only execute a finite number of *J-action*s in each segment of execution — a segment being separated from its follower by the death or the disappearance of some abnormal alive tree.

Definition 10 (Disappear/Die). *Let $\gamma \mapsto \gamma'$ be a 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 $Root(p)$ does not hold in γ'.*
- $Tree(p)$ dies *during the step $\gamma \mapsto \gamma'$ if and only if $Tree(p)$ is alive in γ, yet $Tree(p)$ exists — namely $Root(p)$ holds — and is dead in γ'.*

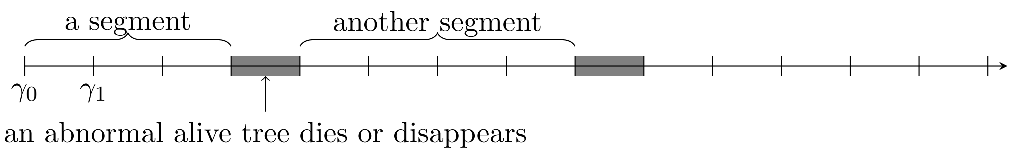


Figure 7: Segments of execution

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

Figure 7 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. □

From Lemma 4, we have the following remark:

Remark 2. *There are at most n steps outside segments (more precisely, the steps where at least one abnormal tree dies or disappears) and these steps necessarily contains an execution of EB-action.*

We now count the number of *J-action*s 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, $AbRoot(p)$ holds in γ, 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. Let T be the tree which contains p in γ. First, T is an abnormal tree in γ. Second, if T does not disappear during the step $\gamma \mapsto \gamma'$, p is still in T in γ' unless T 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 $S\text{-}Trace(KPath(p)) \in EB^*EF^*$, by Observation 1, the status of r in γ is either EF or EB. Hence $AbRoot(r)$ holds in γ: $Tree(r)$ is an abnormal tree in γ.

Assume now that $Root(r)$ holds in γ' (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 *R-action*, its only possible action is *EF-action*. As $Root(r)$ holds in γ', r is still abnormal root in γ'. Let then $q \in KPath(p)$ in γ 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 preserves also $KinshipOk$ 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 a 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 a step and let p be a process such that $AbRoot(p) \wedge p.status = C$ in γ: 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 $SelfRoot(p)$ holds in γ, then $SelfRootOk(p)$ is false in γ, and $SelfRoot(p) \wedge \neg SelfRootOk(p)$ still holds in γ'.

Otherwise, $\neg SelfRoot(p)$ holds in γ, *i.e.*, $p.par \neq p$. Then, $\neg SelfRoot(p)$ still holds in γ'. Let $q \in V$ such that $q = \gamma(p).par$ and consider the following cases:

$\gamma(q).status = EF$**:** Then, $\neg GoodStatus(p, q)$ holds in γ which implies that $\neg KinshipOk(p, q)$ holds in γ. However, $p \in Children_q$ in γ. 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 that the predicate $\neg GoodStatus(p, q)$ holds in γ'. Thus, we have $\neg KinshipOk(p, q)$ in γ'.

$\gamma(q).status = EB$**:** Then, $GoodStatus(p, q)$ holds in γ. So, $AbRoot(p)$ in γ implies that the predicate $\neg GoodIdR(p, q) \vee \neg GoodLevel(p, q)$ holds in γ. 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, $\neg GoodIdR(p, q) \vee \neg GoodLevel(p, q)$ still holds in γ'. So, $\neg KinshipOk(p, q)$ holds in γ'.

$\gamma(q).status = C$**:** As $AbRoot(p)$ holds in γ, $\neg KinshipOk(p,q)$ in γ. Thus, $\neg Allowed(q)$ holds in γ because $p \in Children_q$ and $p.status = C$ in γ. So, q cannot execute *J-action* in $\gamma \mapsto \gamma'$. Then, $\gamma(q).status = C$ and $\gamma(p).status = C$ implies that $GoodStatus(p,q)$ holds in γ. So, $AbRoot(p)$ in γ implies that $\neg GoodIdR(p,q) \vee \neg GoodLevel(p,q)$ holds in γ. As p and q can only modify their status during the step $\gamma \mapsto \gamma'$ (q can only execute *EB-action* in $\gamma \mapsto \gamma'$), $\neg GoodIdR(p,q) \vee \neg GoodLevel(p,q)$ still holds in γ'. So, $\neg KinshipOk(p,q)$ holds in γ'.

In any cases, $\neg KinshipOk(p,q)$ holds in γ'. As the predicate $\neg SelfRoot(p)$ holds in γ', $AbRoot(p)$ holds in γ'. □

Lemma 7. *Let γ be a configuration and let p be a process such that $p.status \in \{EB, EF\}$ in γ. Let T be 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 T 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 $\gamma = \gamma_0\gamma_1...$ 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, $x \in \{0, ..., R-1\}$, 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. □

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

Proof. Let $\gamma \mapsto \gamma'$ be a step. 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.

If $AbRoot(p)$ holds in γ, 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, $\neg AbRoot(p)$, $p.par.status = EB$, and $KinshipOk(p, p.par)$ holds in γ. 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 $\gamma(p).par = \gamma'(p).par$ (p can only change its status to EB in $\gamma \mapsto \gamma'$: either p do not move or executes *EB-action*). 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 $KinshipOk(p, p.par)$ holds in γ, we have that $p \in RealChildren_q$ in γ. Since $\gamma(p).status = C$, q is disabled in γ (because of p) and, as p can only modify its status to EB in $\gamma \mapsto \gamma'$, we still have $p \in RealChildren_q$ in γ', *i.e.*, p and q belong to the same abnormal tree, $Tree(r)$, in γ'. □

Corollary 1. *Let γ be a configuration and let p be a process such that $EBroadcast(p)$ holds in γ. Let T 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, then T 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 $EBroadcast(p)$ holds in γ. 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 $\gamma = \gamma_0\gamma_1...$ 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 from γ_0 and to 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}), and so on.

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

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 $s = \gamma_0\gamma_1\ldots$ 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, some abnormal tree is alive and that in γ_l, this tree is dead or has disappeared. Hence γ_k and γ_l are not in the same segment, a contradiction. □

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-action*s between two *J-action*s 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-action*s 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 be assigned to (1) values which are present in variables idR of processes in A at the first configuration of the segment and (2) to 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 $p_0, \ldots p_{k-1}$ (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-action*s, 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-action*s. Hence, follows:

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. First, by definition, there are at most n abnormal alive trees in the initial configuration. Let $\sharp AbT$ be that number. Moreover, $\sharp AbT$ can only decrease, by Lemma 3.

Let p be a process. We first show that when p executes *R-action* for the first time, $\sharp AbT \leq n-1$. Then, we show that after every subsequent execution of a *R-action* by p, $\sharp AbT$ necessarily decreases. Hence, we will conclude that p cannot execute *R-action* more than n, because $\sharp AbT$ cannot be negative.

Consider the first step $\gamma_i \mapsto \gamma_{i+1}$ where p executes *R-action*. Using Observation 2, $Tree(p)$ exists and is dead in γ_i. Hence, there are at most $n-1$ abnormal alive trees in γ_i.

Consider the $j-th$ execution of *R-action* by p, with $j > 1$. After the $(j-1)-th$ *R-action* of p, the status of p is C. So, between the $(j-1)-th$ and the $j-th$ *R-action*, the status of p thus switches from C to EB and from C to EF, so that p can switch its status from EF to C when executing its $j-th$ *R-action*. Hence, meanwhile there exists a configuration γ_b such that $EBroadcast(q)$ is true and another γ_r, later on such that p executes its $j-th$ *R-action* in $\gamma_r \mapsto \gamma_{r+1}$: Corollary 1 applies and proves that the tree to which p belongs in γ_b is abnormal alive and that tree dies or disappears some step before γ_r, and we are done. □

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 Remark 2, 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 $q.status = EB \wedge (\forall q' \in RealChildren_q, q'.status \neq EB) \wedge p \in KPath(q)$. If $RealChildren_q = \emptyset$, then q can executes *EF-action*. Otherwise, there are two cases. If $\forall q' \in RealChildren_q$ then, $q'.status = EF$ and q can execute *EF-action*. Otherwise, there is $q' \in RealChildren_q$ such that $q'.status = C$ and 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 $q.status = EF \wedge (Root(q) \vee (KinshipOk(q, q.par) \wedge q.par.status \neq EF) \wedge q \in KPath(p)$.

   If $Root(q)$, then $AbRoot(q) \vee SelfRoot(q)$. Now, $q.status = EF$ implies that $AbRoot(q)$ holds. So, in all cases, $q.status = EF \wedge AbRoot(q)$ holds. If $Allowed(q)$ holds, then R-*action* is enabled at q, a contradiction. Otherwise, $\exists r \in Children_q$ such that $\neg KinshipOk(r, q) \wedge r.status = C$. So EB-*action* is enabled at r, a contradiction.

   If $\neg Root(q)$, either $q.par.status = C$, $AbRoot(q)$ holds and we obtain a contradiction as in the case where $Root(q)$ holds, or $q.par.status = EB$ and using the same argument as in case 1, we can deduce that some process is enabled, a contradiction.

   Hence, all 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 be a terminal configuration. Assume first, by contradiction, that there are at least two leaders. As G is connected, $\exists p, q \in V$ such that $Leader(\gamma(p)) \neq Leader(\gamma(q))$ and $q \in \mathcal{N}_p$. Now, 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 $\gamma(p).par = q$. As the level of p is minimum, $\gamma(p).level \leq \gamma(q).level$. 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 3. *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 silent self-stabilizing 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 and we make a worst-case analysis of its stabilization time both in steps and 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 4. *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 5. *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 4. 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 5) 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, $\neg EBroadcast(p) \wedge p.status = C$ holds in γ'.

But $Join(p)$ must be false in γ'. Using Remark 5, 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 5. $\square$

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 $p.idR \neq p$ so $SelfRootOk(p)$ is false in γ. Hence, using Remark 4, the predicate $KinshipOk(p, p.par)$ holds in γ. Since we take p of minimum idR, $p.idR \leq p.par.idR$ in γ. $GoodIdR(p, p.par)$ implies that $p.idR \geq p.par.idR$, so $p.idR = p.par.idR$. 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 6. *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 disabled forever.*

Proof. Let γ be a clean configuration. Let p be a process with $\gamma(p).idR = \ell$. By Remark 5, only *J-action* can be enabled in γ. Moreover, its guard reduces to $\exists q \in \mathcal{N}_p, (q.idR < p.idR)$. But Lemma 15 ensures that this cannot be true, hence p is disabled in γ. Then, by Lemma 13 and Remark 6, this will be true forever. □

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 ith 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 $\|p, \ell\| = 0$, $p = \ell$. Notice that if the predicate $GoodIdR(p, p.par)$ holds in ρ_0, it would implies that $p.idR < p$ which is false by Lemma 15. So $KinshipOk(p, p.par)$ cannot hold in ρ_0. Hence, $SelfRoot(p) \wedge SelfRootOk(p)$ holds in ρ_0 (by Remark 4) 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. □

Dirty Configurations In the previous paragraph, we proved that, starting from a clean initial configuration, 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 the predicate $EBroadcast(p)$ 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, $AbRoot(q) \vee SelfRoot(q)$ holds in γ. 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.

□

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 any abnormal 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 that the predicate $AbRoot(p)$ 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 then, $GoodStatus(p, p.par)$ is false in γ', or $p.par$ 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, the predicate $GoodStatus(p, p.par)$ is false in γ', which implies that $EBroadcast(p)$ holds in γ'. □

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

Proof. Consider any execution $e = \gamma_0, \ldots$. $\forall i > 0$, we denote by γ_{R_i} the last configuration of the ith round and so the first configuration of the $i+1$th round of e. Moreover, let $\gamma_{R_0} = \gamma_0$ be the initial configuration. We show by induction on the length of the $KPath$s that, $\forall i \geq R_d$ ($d \geq 1$), $\forall p \in V$, if p is in an abnormal tree and $|KPath(p)| \leq d$ in γ_i, then p is dead in γ_i.

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_0} and if predicate ($p.status = C \wedge AbRoot(p)$) becomes false in some configuration, 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 (that is, from γ_{R_1}).

By definition, *EB-action* is enabled at p in γ_{R_0} and p executes *EB-action* during the first round (using Lemma 18). Hence, p is dead at the end of the first round, and we are done.

Induction Hypothesis: Let $d \geq 1$. Assume that $\forall i \geq R_d$, $\forall p \in V$, if p belongs to an abnormal tree and $|KPath(p)| \leq d$ in γ_i, then p is dead in γ_i.

Induction Step: We first show that for every $p \in V$, for every $i \geq R_d$, if ($p.status = C \wedge |KPath(p)| \leq d+1$) is false in configuration γ_i, then for every $j \geq i$, ($p.status = C \wedge |KPath(p)| \leq d+1$) is false in configuration γ_j.

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

- If $\gamma_j(q).status = EB$, then p does not execute any action during the step $\gamma_j \mapsto \gamma_{j+1}$ (otherwise, $\gamma_{j+1}(p).status \neq C$ or $\gamma_{j+1}(p).par \neq q$). Hence, $\gamma_j(p).status = \gamma_{j+1}(p).status = C$. By hypothesis, "$p.status = C \wedge |KPath(p)| \leq d+1$" is false in γ_j, so we have $|KPath(p)| > d+1$ in γ_j.
  Now, $\gamma_j(p).status = C$ and $\gamma_j(q).status = EB$, so $S\text{-}Trace(KPath(p)) = EB^+C$ in γ_j (Observation 1) and p is the only process in its $KPath$ that can execute an action in $\gamma_j \mapsto \gamma_{j+1}$. Hence, for every q such that $q \in KPath(p)$ in γ_j, $q \in KPath(p)$ in γ_{j+1}, and then $|KPath(p)| > d+1$ in γ_{j+1}. So $p.status = C \wedge |KPath(p)| \leq d+1$ is false in γ_{j+1}, a contradiction.
- If $\gamma_j(q).status = C$, then q is in an alive abnormal tree in γ_j (indeed, q executes *EB-action* in $\gamma_j \mapsto \gamma_{j+1}$, and so Lemma 8 applies). As q is alive in γ_j, we have $|KPath(q)| > d$ in γ_j 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_j is EB.
  Now, $|KPath(q)| > d$ and $S\text{-}Trace(KPath(q)) = EB^+C$ in γ_j (Observation 1). So, q is the only one in its $KPath$ that executes an action in $\gamma_j \mapsto \gamma_{j+1}$ and this action is *EB-action*, that maintains the $KinshipOk$ relation. Hence, $|KPath(q)| > d$ in γ_{j+1} and consequently, $|KPath(p)| > d+1$ in γ_{j+1}, a contradiction.

Hence, $\forall p \in V$, if ($p.status = C \wedge |KPath(p)| \leq d+1$) is false in some configuration γ_i with $i \geq R_d$, then ($p.status = C \wedge |KPath(p)| \leq d+1$) remains false forever.

Now, *EB-action* is continuously enabled $\forall p$ such that p is alive $|KPath(p)| = d+1$ in γ_{R_d} (by induction hypothesis and Lemma 18). So, p becomes dead during the round and, $\forall j \geq R_{d+1}$, γ_j contains no alive process p such that $|KPath(p)| \leq d+1$.

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

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

Proof. Let $\gamma \mapsto \gamma'$ be a step. Assume by contradiction *EF-action* is enabled at a process p in γ and is not enabled in 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 $EFeedback(p)$ holds in γ, $\gamma(p).status = \gamma'(p).status = EB$. As $EFeedback(p)$ 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. □

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 $e = \gamma_0, \ldots$ starting from a configuration that contains abnormal trees and where all abnormal trees are dead. $\forall i > 0$, we denote by γ_{R_i} the last configuration of the ith round and so the first configuration of the $i+1$th round. Moreover, let $\gamma_{R_0} = \gamma_0$ be the initial configuration.

Claim 1: $\forall p \in V$, $\forall i \geq R_0$, if $\gamma_i(p).status \neq EB$, 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 to Lemma 3.

In any configuration γ, we denote by $MaxLengthKPath(p) = \max\{|KPath(q)|, q \in V \wedge p \in KPath(q)\}$. Again in γ, we define $L(p) = MaxLengthKPath(p) - |KPath(p)|$ and $EBL(p,k) \equiv p.status = EB \wedge L(p) = k$.

Claim 2: $\forall i \geq R_0$, if $EBL(p, k_i)$ holds in γ_i, then $\forall j \geq i, \forall k_j < k_i, \neg EBL(p, k_j)$ holds in γ_j.

> If $j = i$, $EBL(p, k_j)$ is false for $k_j < k_i$ because $L(p)$ cannot have two different values in a same configuration. Assume now $j > i$. 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 $k_j < k_i$. So, $\gamma_i(p).status = \gamma_j(p).status = EB$ and 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 be any process such that $p \in KPath(q)$ in γ_i. So, $KPath(q) = q_0 \ldots q_i \ldots q_k$ where $q_i = p$ and $q_k = q$, and $S\text{-}Trace(KPath(q)) = EB^+EF^*$ in γ_i. There is a unique process in $KPath(q)$ that can execute an action in $\gamma_i \mapsto \gamma_{i+1}$ (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 γ_i, $q' \in KPath(q)$ in γ_{i+1}. We can apply this latter property to every process r such that $p \in KPath(r)$ and $|KPath(r)| = MaxLengthKPath(p)$ in γ_i: $p \in KPath(r)$ in γ_{i+1} and the value of $|KPath(r)|$ in γ_{i+1} is greater than or equal to the value of $|KPath(r)|$ in γ_i. So, $EBL(p, k_{i+1})$ holds with $k_{i+1} \geq k_i$. Applying the same argument on step $\gamma_{i+1} \mapsto \gamma_{i+2}$, *etc.*, until step $\gamma_{j-1} \mapsto \gamma_j$, we obtain 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 $\gamma_{x+1}(p).status = EB$. This contradicts Claim 1.

We show by induction that $\forall i \geq R_d$ with $d \geq 1$, $\forall p \in V$, $\forall k \leq d-1$, $EBL(p,k)$ is false in γ_i.

Base case: There are three cases:

1. If $L(p) = 0$ in γ_{R_0} and $\gamma_{R_0}(p).status = EB$, then EF-*action* is enabled at p in γ_{R_0}, p executes EF-*action* during 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_i, $\forall i \geq R_1$.
2. If $\gamma_{R_0}(p).status \neq EB$, $p.status \neq EB$ forever (Claim 1) and then $EBL(p, 0)$ is false forever.
3. If $EBL(p,k)$ holds in γ_{R_0} with $k > 0$, $EBL(p, 0)$ is false forever (Claim 2).

Induction hypothesis: $\forall i \geq R_d$ with $d \geq 1$, $\forall p \in V$, $\forall k \leq d-1$, $EBL(p,k)$ is false in γ_i.

Induction step: There are four cases:

1. If $L(p) = d$ and $\gamma_{R_d}(p).status = EB$, then $\forall q \in RealChildren_p$ in γ_{R_d}, $L(q) < d$ by definition and $\gamma_{R_{d(q)}}.status \neq EB$ by induction hypothesis. Now, the trees are dead, so $\gamma_{R_d}(q).status = EF$. Hence, EF-*action* is enabled at p in γ_{R_d}. By Lemma 19, p executes EF-*action* during the round and gets status EF. Then, $p.status \neq EB$ forever (Claim 1), so $EBL(p, d)$ is false at the end of the $d + 1$th round and remains false forever.
2. If $L(p) = d$ and $\gamma_{R_d}(p).status \neq EB$, then, using Claim 1, $p.status \neq EB$ forever. So, $EBL(p, d)$ is false forever.
3. If $L(p) < d$, $\gamma_{R_d}(p).status \neq EB$ by induction hypothesis and we conclude as in case 2.
4. If $EBL(p, k)$ holds in γ_{R_d} with $k > d$, $EBL(p, i)$ is false forever $\forall i \leq d$ (Claim 2).

With $d = n$, we have $\forall i \geq R_n$, $\forall p \in V$, $\forall k \leq n - 1$, $EBL(p, k)$ is false in γ_i: hence, in at most n rounds, there is no more process of status EB in abnormal trees, those ones being dead. So, all processes (and in particular the abnormal roots) in 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.
  1. 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 then $AbRoot(p)$ 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'.
  2. 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 so $\gamma(p).idR = \gamma(q).idR$ but $\gamma(p).level \neq \gamma(q).level + 1$. First, 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* and consequently $\gamma'(q).status = C$. 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 said to be* abnormal *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, by Lemma 3, no alive abnormal tree can be created, we have the following remark:

Remark 7. *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 7, 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 status C. So, by Observation 1, we have:

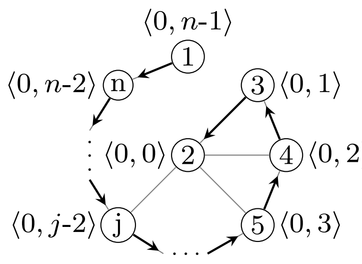
Remark 8. *Every process has status C in a configuration containing no abnormal processes. Moreover, this configuration is clean.*

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

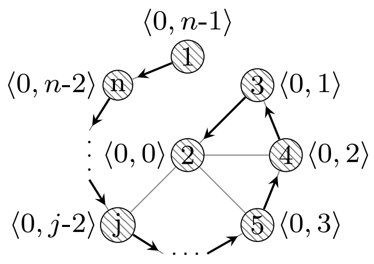
Theorem 8. *In at most $3n$ rounds, the system reaches a clean configuration.*

Then, using Theorems 4 and 8 we get:

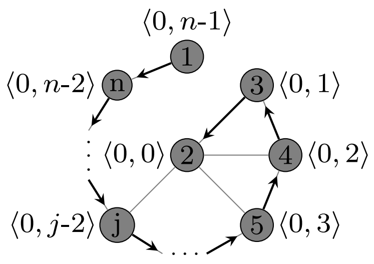
Theorem 9 (Round Complexity)**.** *In at most $3n + \mathcal{D}$ rounds, the system reaches a terminal configuration.*



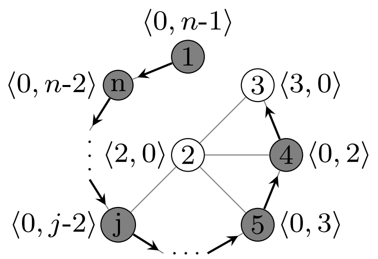
(a) The initial configuration. $\{p_2, p_j\}$ is the "last" edge to p_2 ($j = k + 3$).



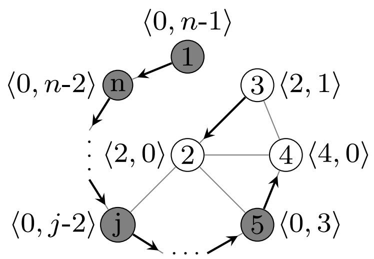
(b) In n rounds, the EB-wave reaches p_1.



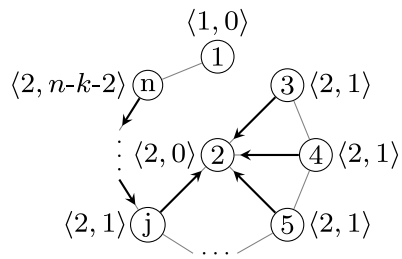
(c) In n rounds, p_2 gets status EF.



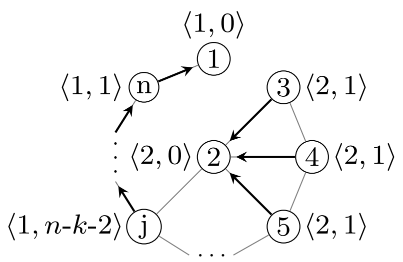
(d) p_2 and p_3 sequentially execute *R-action*.



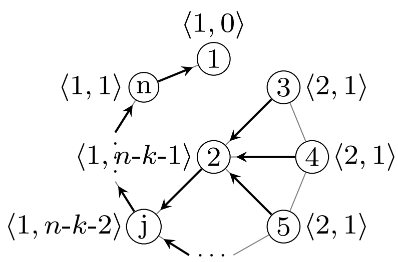
(e) p_3 executes *J-action* and p_4 simultaneously executes *R-action*.



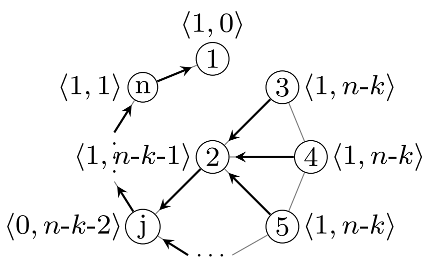
(f) In $n - 3$ rounds, the cleaning is finished.



(g) In $n - k - 2$ rounds, processes n to $k - 3$ joins $Tree(1)$.



(h) Processes p_2 and p_{k-2} simultaneously execute *J-action*.



(i) In one round, the system reaches a terminal configuration where p_1 is the leader.

Figure 8: An example in $3n + \mathcal{D}$ rounds

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, for $i \in \{1, \dots, n\}$. Figure 8a shows the system in its initial configuration. In details, processes p_1, p_n, ...,p_2 form a chain, *i.e.*, $\{p_1, p_n\} \in E$ and $\{p_i, p_{i-1}\} \in E$, $\forall i \in \{3, \dots, n\}$.

We add k edges to p_2, with $2 \leq k \leq n-2$, as follows:

If $k = n - 2$, $\{p_2, p_1\} \in E$ and for $\forall i \in \{4, \dots, n\}$, $\{p_2, p_i\} \in E$,

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

Notice that the diameter of the network is $n - k$ and can be adjusted by adding or removing some edges to p_2.

We assume the following initial configuration:

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

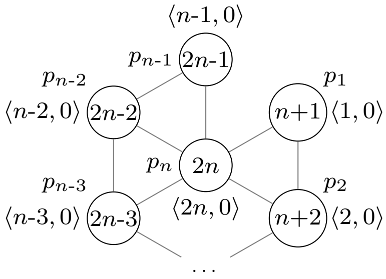
We consider a synchronous daemon, *i.e.*, in a configuration γ, every process in $Enabled(\gamma)$ is selected by the daemon to execute an action. So, in this case, every round lasts exactly one step.

The execution is then as follows:

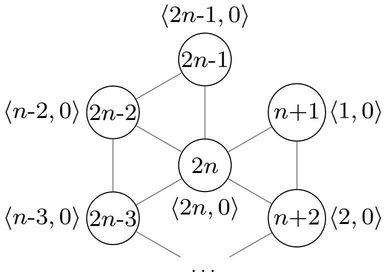
- $p_2, p_3, p_4, \dots, p_n, p_1$ sequentially execute *EB-action*: n rounds. (See Figure 8b.)
- $p_1, p_n, p_{n-1}, \dots, p_2$ sequentially execute *EF-action*: n rounds. (See Figure 8c.)
- p_2 and p_3 sequentially execute *R-action*: 2 rounds. (See Figure 8d.)
- For $i = 4, \dots, n$, simultaneously 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 8e and 8f.)
- 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 8g.)
- p_2 and p_{k+2} simultaneously execute *J-action* to join $Tree(1)$: 1 round. (See Figure 8h.)
- $p_3, \dots, p_{k+1}$ simultaneously execute *J-action* and then the configuration is terminal: 1 round. (See Figure 8i.)

Hence, the execution lasts exactly $3n + (n - k) = 3n + \mathcal{D}$ rounds. Using Theorem 9 we can conclude:

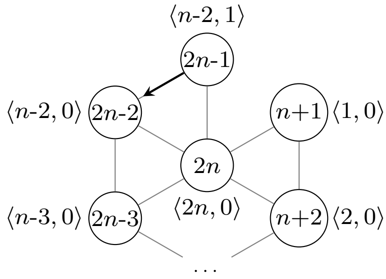
Theorem 10. *In the worst case, the round complexity of $\mathcal{LE}$ is exactly $3n + \mathcal{D}$ rounds.*



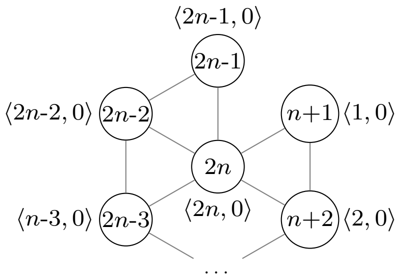
(a) The initial configuration



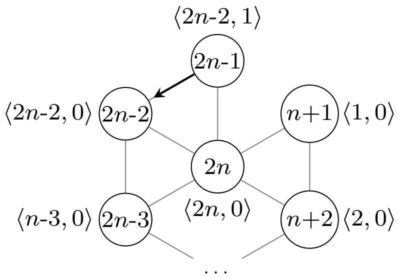
(b) In three steps, p_{n-1} becomes normal



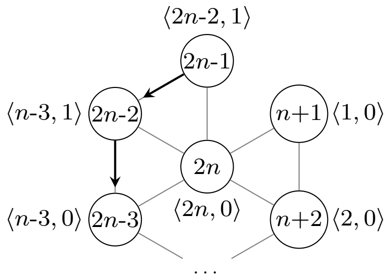
(c) p_{n-1} executes *J-action* and joins $Tree(p_{n-2})$



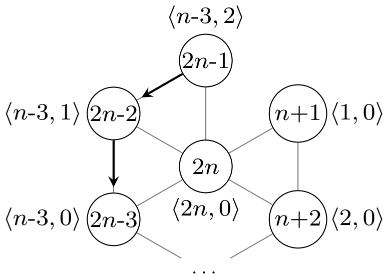
(d) In six steps, the abnormal tree rooted in p_{n-2} is cleaned



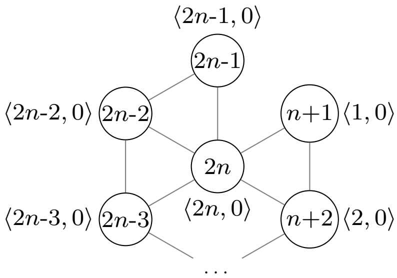
(e) p_{n-1} executes *J-action* and joins the normal tree $Tree(p_{n-2})$



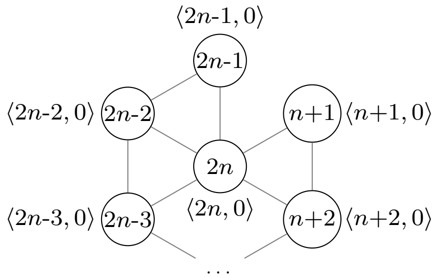
(f) p_{n-2} executes *J-action* and joins the abnormal tree $Tree(p_{n-3})$



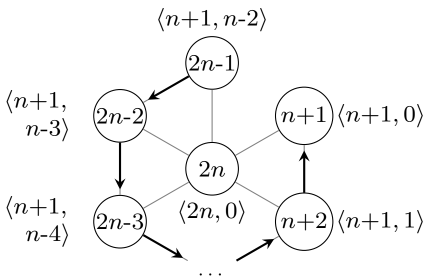
(g) p_{n-1} executes *J-action* and updates its idR to $n-3$.



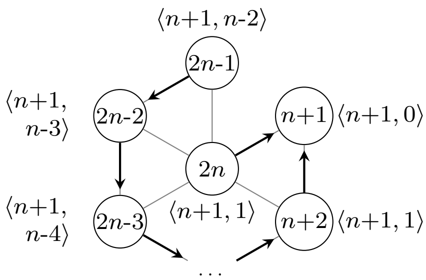
(h) In nine steps, the abnormal tree rooted in p_{n-3} is cleaned



(i) There is no more abnormal trees



(j) In $\sum_{j=1}^{n-2} j$ steps, processes p_{n-1} to p_2 elect p_1



(k) In one step, the system reaches a terminal configuration where p_1 is leader.

Figure 9: An example in $\Omega(n^3)$ steps

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 $V = \{p_1, \ldots, p_n\}$ such that p_i has ID $n + i$, $\forall i \in \{1, \ldots, n\}$. Figure 9a shows the network in this initial configuration. In details, there are $2n-3$ edges: $\{p_i, p_{i+1}\}$ for $i \in \{1, \ldots, n-2\}$ and $\{p_i, p_n\}$ for $i \in \{1, \ldots, n-1\}$. (Notice that the diameter of this network is 2.) The initial configuration is as follows:

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

We consider the following execution:

- For $i = n-1, n-2, \ldots, 1$, we clean $Tree(p_i)$ the following way:
  1. If $i \leq n-2$, for $j = n-2, n-1, \ldots, i$,
     (a) For $k = j+1, j+2, \ldots, n-1$, p_k joins $Tree(p_j)$.
     Case 1 lasts $\sum_{j=1}^{n-1-i} j = (n-i-1)(n-i)/2$ steps.
  2. $p_i, p_{i+1}, \ldots, p_{n-1}$ sequentially execute *EB-action*: $n-i$ steps.
  3. $p_{n-1}, p_{n-2}, \ldots, p_i$ sequentially execute *EF-action*: $n-i$ steps.
  4. $p_i, p_{i+1}, \ldots, p_{n-1}$ sequentially execute *R-action*: $n-i$ steps.

  Figures 9e to 9h 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: $\sum_{j=1}^{n-2} j = (n-1)(n-2)/2$ steps (Figure 9j).
- p_n executes *J-action* to join $Tree(p_1)$: 1 step (Figure 9k).

Hence, the complete execution lasts:

$$3 + \sum_{i=1}^{n-2} \left(3(n-i) + \frac{(n-i-1)(n-i)}{2} \right) + \frac{(n-1)(n-2)}{2} + 1 = \frac{n^3}{6} + \frac{3}{2}n^2 - \frac{8}{3}n + 2 \text{ steps}$$

So, there exists an execution in $\Omega(n^3)$. Using Theorem 3, we can conclude:

Theorem 11. *In the worst case, the step complexity of $\mathcal{LE}$ is in $\Theta(n^3)$ steps.*

5 Step Complexity of Algorithm $\mathcal{DLV}$

In this section, we study the step complexity of the algorithm given in [10], called here $\mathcal{DLV}$.[2] 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 main principles in Subsection 5.1. Then, we give in Subsection 5.2 an example of a class of network in which there is a possible execution that stabilizes in $\Omega(n^4)$ steps. Finally, in Subsection 5.3, we generalize the previous example to a class of network where there is a possible execution that stabilizes in $\Omega(n^\alpha)$ for any $\alpha \geq 4$.

Algorithm 2 Algorithm $\mathcal{DLV}$ [10] for every process p

Variables

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

Macros

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

Predicates

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

Guards

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

Actions

$$
\begin{array}{llcll}
J & \text{(priority 1)} & :: & Join(p, q) & \rightarrow \; p.key \leftarrow SuccKey(q); p.parent \leftarrow q; \\
 & & & & \quad\; p.color \leftarrow 1; p.done \leftarrow \texttt{false}; \\
R & \text{(priority 2)} & :: & Reset(p) & \rightarrow \; p.key \leftarrow SelfKey(p); p.parent \leftarrow p; \\
 & & & & \quad\; p.color \leftarrow 2; p.done \leftarrow \texttt{false}; \\
C1 & \text{(priority 3)} & :: & Color1(p) & \rightarrow \; p.color \leftarrow 1; p.done \leftarrow Done(p); \\
C2 & \text{(priority 3)} & :: & Color2(p) & \rightarrow \; p.color \leftarrow 2; p.done \leftarrow Done(p); \\
UD & \text{(priority 4)} & :: & UpdateDone(p) & \rightarrow \; p.done \leftarrow Done(p);
\end{array}
$$

(a) 7 can execute $C2$ action and get color 2.

(b) 7 can execute $C1$ action and get color 1.

Figure 10: Guards of color actions. The ID is represented inside the node. The label next to a node shows its *key*. The arrows represent *parent* pointers. No arrow exits a node if its parent is itself. The filling represents the color: gray for 1 and white for 2.

5.1 Overview of $\mathcal{DLV}$

The formal code of Algorithm $\mathcal{DLV}$ is given in Algorithm 2. This algorithm uses priorities. Each of its actions is given with priority number. When an enabled process is selected by the daemon, it only executes its enabled action with the lowest priority number.

Algorithm $\mathcal{DLV}$ elects the process of minimum ID, ℓ, and builds a breadth-first spanning tree rooted at ℓ. To ensure that every process knows which one is elected, it maintains a variable *leader* in which is saved the alleged leader. Variables *parent* and *level* are used to define the tree. The *key* of a process p is the combination of its two variables $p.leader$ and $p.level$. Notice that the keys are ordered using the classical lexical order.

Let p be a process. Let q be its neighbor of smallest key ($BestNbrKey(p)$). Suppose the key of process p is not the immediate successor of the q's key or $p.parent \neq q$. p may execute Action J to modify its *key* and its *parent* pointer accordingly. Notice that, contrary to our algorithm, p can execute Action J and change its parent when there is a neighbor with the same leader but with a level smaller than $p.level - 1$, in order to build a breadth-first spanning tree. Note also that the execution of Action J is constrained by the use of a *color*, whose goal will be explained later.

As in $\mathcal{LE}$, Datta *et al* define a "good relation" between a process p and its parent: $IsTrueChld(p)$. This 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 the $IsTrueChld$ relation defines a tree. The root of a tree can be a *true root* ($IsTrueRoot(p)$), *i.e.*, the key of p is a self key ($\langle p, 0\rangle$). In this case, the tree is said to be *normal*. Otherwise, the root is a *false root* ($IsFalseRoot(p)$), *i.e.*, neither a true root nor a true child, and the tree is said to be *abnormal*.

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*, whose value is either 1 or 2. A process can only choose as parent a neighbor of color 2 and after executing Action J, the process gets color 1.

A process can change its color, by executing Actions $C1$ or $C2$, if it has the same color than its parent (it is trivially satisfied for every true root) and if all of its true children have the other color (see Figure 10). There is an additional constraint to change a color to 1: as a process cannot recruit when it has color 1, a process p of color 2 must not change its color while it can recruit processes (while $Recruits(p) \neq \emptyset$).

[2] $\mathcal{DLV}$ stands for "Datta, Larmore, and Vemula."

To add a new level in the tree, the leaves must change their color to 2. Then, the goal is to propagate up in the tree a first wave of Actions $C1$ initiated by the parents of the leaves, so that a second wave of Actions $C2$ can be initiated by the leaves. To ensure that, the root should absorbed a (previous) wave. But, only a true root can absorb a color wave. Indeed, the priorities on actions prevent a false root to change its color (before it resets) and, so, to absorb a color wave.

Therefore, the colors of the processes in an abnormal tree eventually alternate, *i.e.*, the parents and their true children do not have the same color, and no more process can join the tree: the tree is *color locked*. Then, the false root eventually resets by executing Action R, and so forth. Once all abnormal trees have been removed, ℓ is a true root and regularly absorb color waves allowing then the leaves of its tree to recruit processes.

Finally, in $O(n)$ rounds, ℓ is elected and a breadth-first spanning tree rooted at ℓ is built. Notice that the color waves might never end. So, an additional mechanism allow to ensure the silence by using a Boolean variable *done* and Action UD. When a process p believes that the construction of the final tree is finished (because it cannot recruit processes anymore) and all its true children q (if any) have set their variables $q.done$ to `true`, $p.done$ is set to `true`. Moreover, a true root r cannot change its color once $r.done$ holds. In this case, we said that r is *color frozen*. Thus, after the completion of the final tree construction, the value `true` is propagated up in the tree into the *done* variables, and in $O(\mathcal{D})$ rounds, the system reaches a terminal configuration.

Example of execution Figure 11 shows an example of execution of $\mathcal{DLV}$ (for sake of simplicity, we do not consider the *done* variables and Actions UD in this example). In the initial configuration (Configuration a), the leader of process 7 is 1, the only fake id. Moreover, 5 has already chosen 7 as parent. Then, in step a $\mapsto$ b, 2 and 3 execute Action J and choose 7 (of color 2) as parent. Note also that 5 has the same color than its parent (7), has no true child, and cannot recruit any other process. So 5 executes Action $C1$ and gets color 1 in b $\mapsto$ c. No more process can join the tree rooted at 7 and the tree is color locked (7 is a false root and cannot change its color), so 7 resets during c $\mapsto$ d. In Configuration d, 2, 3 and 5 are false roots. In d $\mapsto$ e they execute Action R in turns. Then, in e $\mapsto$ f, processes 4, 5, 6, 7, and 8 execute Action J to choose 2 as parent. In Configuration f, 3 cannot join the tree rooted at 2 because all its neighbors have color 1. 2 changes its color to 1 by executing Action $C1$ in f $\mapsto$ g. Then, processes 4, 5, 6, 7, and 8 get color 2 by executing Action $C2$ in g $\mapsto$ h. Finally, 3 is allowed to execute Action J and joins the tree rooted at 2 in h $\mapsto$ i.

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

First, we show an execution of $\mathcal{DLV}$ that lasts $\Omega(n^4)$ steps.

Network and initial configuration We consider a network made of $n = L \times \beta$ processes with $L = 8$ and $\beta \geq 2$: $p_{(1,1)}, p_{(1,2)}, \dots, p_{(1,\beta)}, p_{(2,1)}, p_{(2,2)}, \dots, p_{(2,\beta)}, \dots, p_{(8,1)}, p_{(8,2)}, \dots, p_{(8,\beta)}$ such that the ID of $p_{(i,j)}$ is $(i-1)\beta + j, \forall i \in \{1, \dots, 8\}, \forall j \in \{1, \dots, \beta\}$. Notice that 0 is a fake ID smaller than every ID in the network.

Figure 12a shows the structure of the network and the initial configuration. In details, the processes form β columns: $\forall i \in \{2, \dots, 8\}, \forall j \in \{1, \dots, \beta\}, \{p_{(i-1,j)}, p_{(i,j)}\} \in E$.

There are also three complete bipartite subgraphs: $\forall j, j' \in \{1, \dots, \beta\}, j' \neq j$,

$$\{p_{(4,j)}, p_{(5,j')}\} \in V, \{p_{(6,j)}, p_{(7,j')}\} \in E \text{ and } \{p_{(7,j)}, p_{(8,j')}\} \in E$$

These bipartite subgraphs split the network in four layers:

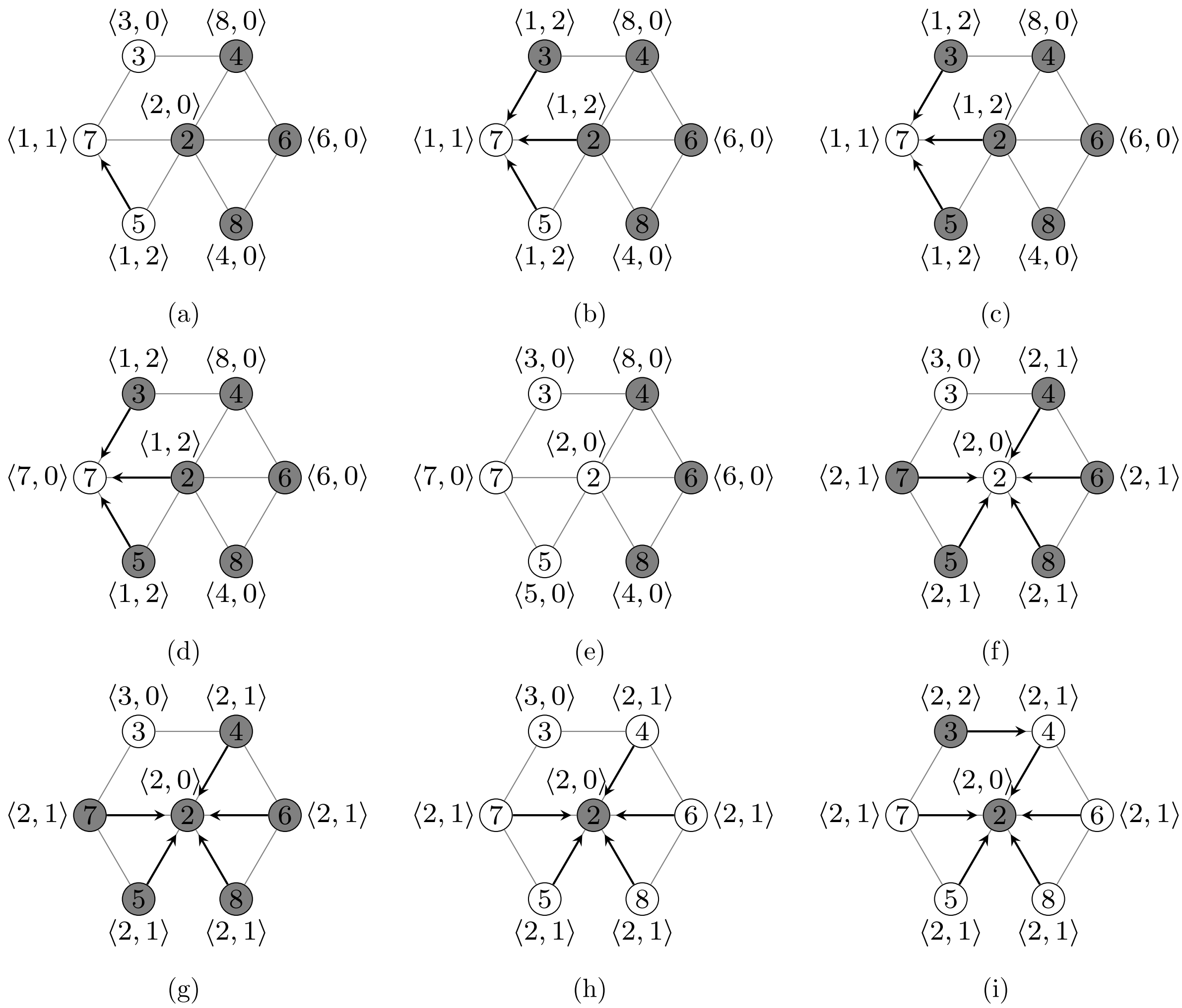


Figure 11: Example of execution of algorithm $\mathcal{DLV}$.

- Layer 1: line 8
- Layer 2: line 7
- Layer 3: lines 5 and 6
- Layer 4: lines 1 to 4

We choose the following initial configuration.

- For $i \in \{1, \ldots, 8\}$, for $j \in \{1, \ldots, \beta\}$, $p_{(i,j)}.leader = 0$, $p_{(i,j)}.level = i$ and $p_{(i,j)}.done =$ `false`
- For $j \in \{1, \ldots, \beta\}$,
  - $p_{(1,j)}.parent = p_{(1,j)}$
  - $p_{(5,j)}.parent = p_{(4,1)}$
  - $p_{(7,j)}.parent = p_{(6,1)}$
  - $p_{(8,j)}.parent = p_{(7,1)}$
  - For $i \in \{2, 3, 4, 6\}$, $p_{(i,j)}.parent = p_{(i-1,j)}$
- For $i \in \{1, \ldots, 8\}$, $p_{(i,1)}.color = (i \mod 2) + 1$
- For $j \in \{2, \ldots, \beta\}$,
  - $p_{(8,j)}.color = 1$
  - For $i \in \{1, \ldots, 7\}$, $p_{(i,j)}.color = 2$

Overview of the execution We first give an illustrative execution to understand the $\Omega(n^4)$ lower bound.

We start with Configuration a of Figure 12. Starting from this configuration, all the processes of the first column and of the last line successively reset. We obtain configuration b. This costs at least β steps (since the reset of the last line can be sequential). Then, all processes $p_{(8,.)}$ can join $p_{(7,2)}$ (which has the fake id 0 as *leader*). This leads to Configuration c. Then, we can reset $p_{(7,2)}$ and the last line (at least β steps). Again processes $p_{(8..)}$ can join $p_{(7,3)}$ and we can reset, *etc.*, until we reset $p_{(7,\beta)}$ and the last line to obtain Configuration d. Overall, this costs at least β^2 steps.

From Configuration d, we can rebuilt the tree on $p_{(6,2)}$. The tree is shown in Configuration e, and we can reset the processes following the order given by the arrow in Configuration e. We obtain Configuration f. Again we can start the succession of buildings and resets bottom-up just as before, but this time, until resetting a tree rooted at $p_{(5,\beta)}$ (Configuration g). This costs at least β^3 steps.

From Configuration g, we can rebuild a tree on the second column until reaching Configuration h . This latter is similar to the first one, Configuration a. The only difference is that the main tree is now rooted at $p_{(1,2)}$ instead of $p_{(1,1)}$. We can repeat the same scheme on each column. This leads to an execution of at least β^4 steps.

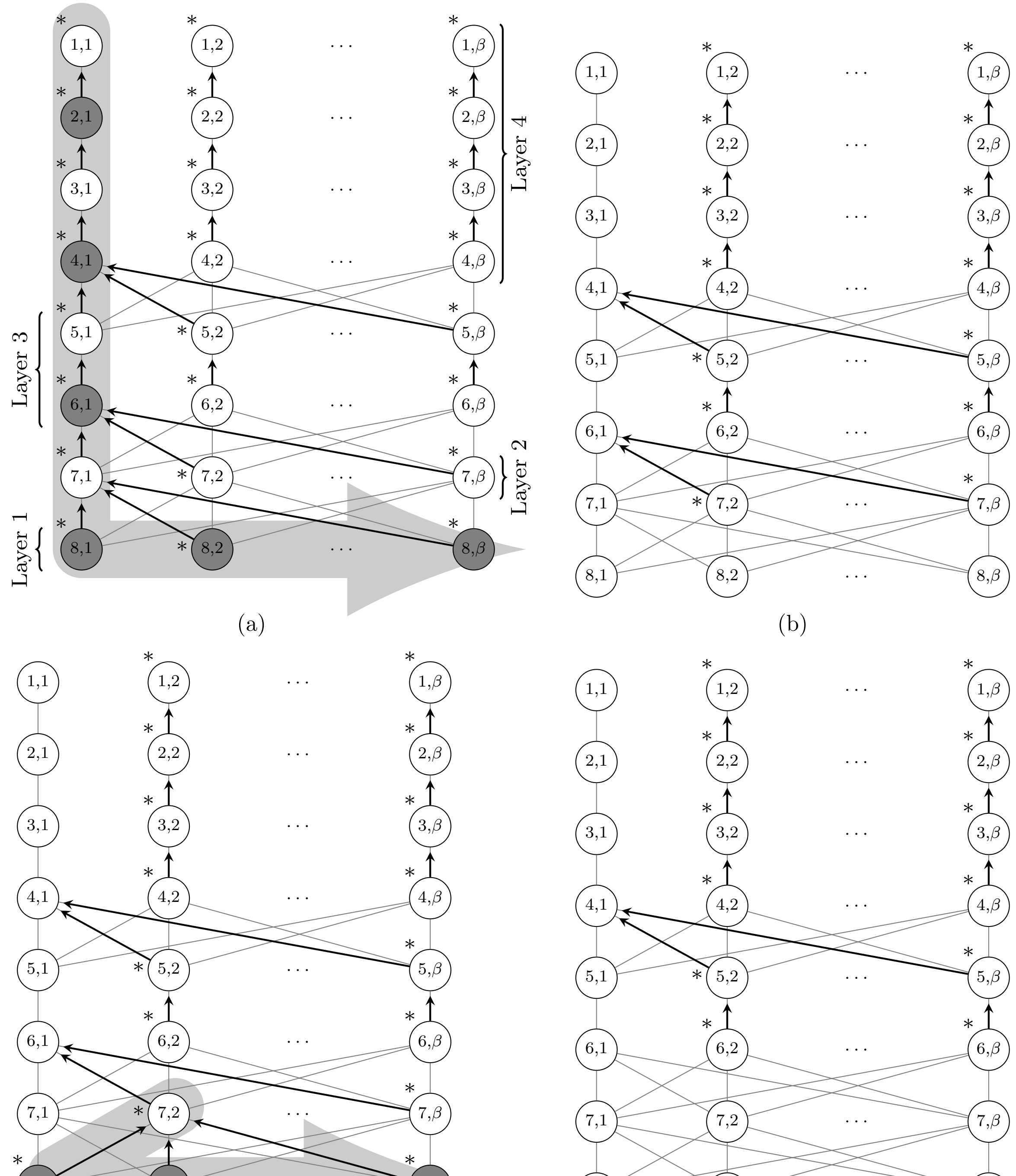
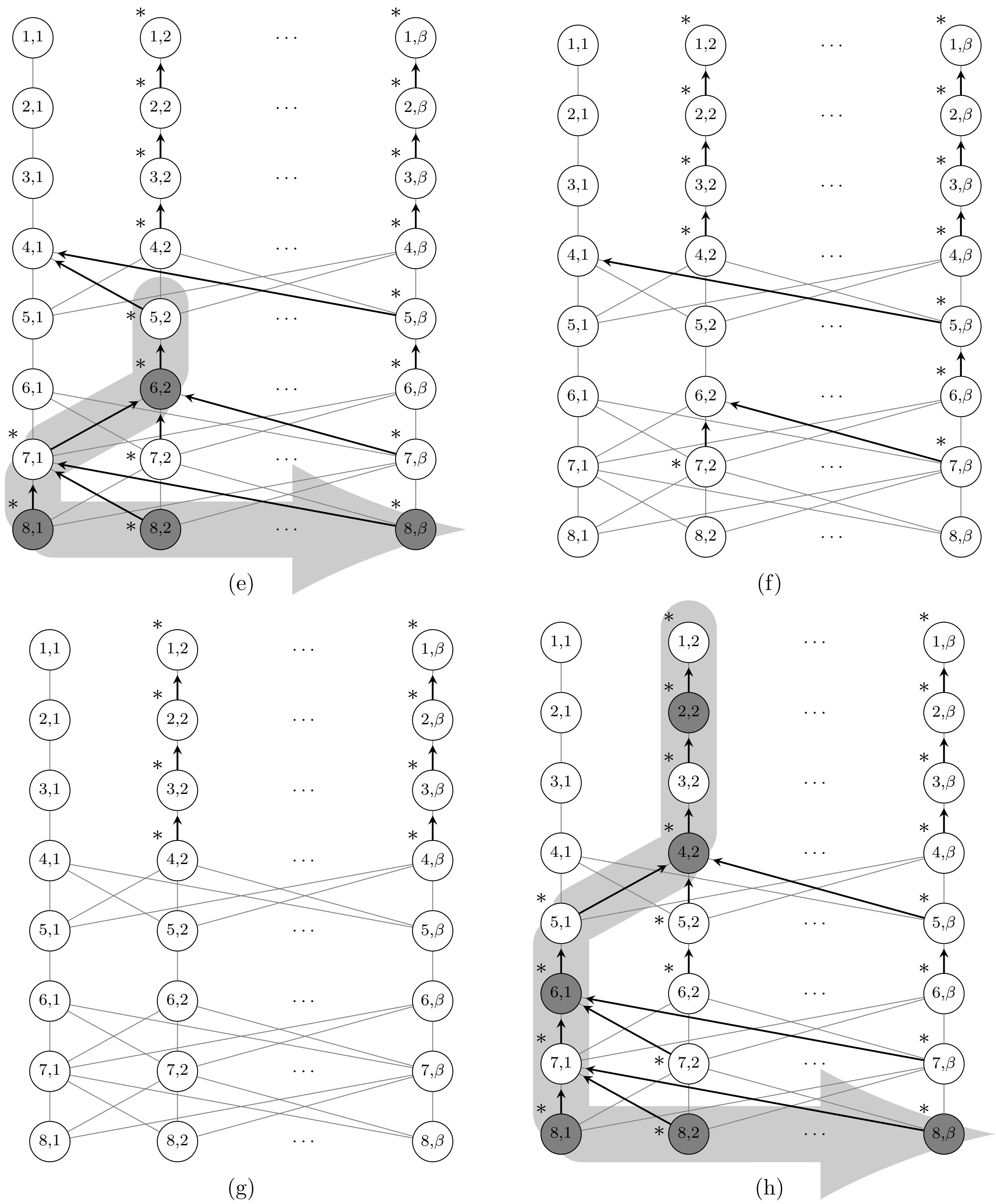


Figure 12: Intuitive idea of the execution. The *leader* of a process is 0 if the process is labeled with a star, its own ID otherwise. *level* is not represented as it is always correct. The plain gray arrows show the processes that successively reset.



(e) (f)

(g) (h)

Figure 12: (continued)

Execution in details Now, let see the details of the execution. We consider an unfair daemon which selects the enabled processes according to the function DAEMON given in Algorithms 3 and 4. In this algorithm, $top(i)$ (respectively $bottom(i)$) is the number of the first line (respectively last line) of layer i. More precisely:

$$top(i) = L - 2^{i-1} + 1$$

$$bottom(i) = \begin{cases} top(1) & \text{if } i = 1 \\ top(i-1) - 1 & \text{if } i > 1 \end{cases}$$

In BUILD$(layer, column)$, all the processes of lines $top(layer)$ to 8 execute line by line Action J. Notice that every process of line $top(layer)$ chooses the process $p_{(top(layer)-1, column)}$ as parent.

In RESET$(layer, column)$, all the processes on column $column$ from the one on line $top(layer+1)$ to the one on line $bottom(layer+1)$ execute Action R (except for layer 1 where all the processes of line 8 also execute Action R). Then, RESET$(layer-1, i)$ and BUILD$(layer-1, i+1)$ are called for each column $i = 1, \ldots, \beta - 1$. Finally, RESET$(layer-1, \beta)$ is executed.

We count how many times processes $p_{(8,.)}$ executes Action R:

- Each process $p_{(8,.)}$ executes once Action R on line 15 of Algorithm 3 in function RESET$(layer, column)$, when $layer = 1$: at least β processes execute Action R.
- RESET$(3, column)$ is called β times by function DAEMON.
- RESET$(2, column)$ is called β times by function RESET$(3, column)$.
- RESET$(1, column)$ is called β times by function RESET$(2, column)$.

Hence, Action R is executed β^4 times by the processes of line 8. Now, $\beta = n/8$. Hence we can conclude:

Theorem 12. *For every $\beta \geq 2$, there exists a network of $n = 8 \times \beta$ processes in which there exists a possible execution that stabilizes in $\Omega(n^4)$ steps.*

5.3 Generalization to an Example in $\Omega(n^\alpha)$ Steps

We note E_4 the graph built for the example in $\Omega(n^4)$ steps and shown in Figure 12a. Then, starting from $E_{\alpha-1}$ ($\alpha \geq 5$), we can build E_α, a graph for which there exists an execution in $\Omega(n^\alpha)$ steps. The construction is based on the same principle as in Subsection 5.2, by adding a layer. If $E_{\alpha-1}$ has $L\beta$ processes $p_{(i,j)}$ ($1 \leq i \leq L$, $1 \leq j \leq \beta$), then E_α 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:

1. We increase the *level* and the ID of the $L\beta$ processes of $E_{\alpha-1}$ as follows: $\forall i \in \{1, \ldots, L\}$, $\forall j \in \{1, \ldots, \beta\}$, $q_{(i+L,j)} = p_{(i,j)}$. 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 *color* and *done* do not change. If $i \neq 1$, the *parent* remains the same. Otherwise, see step 3.
2. At the top of $E_{\alpha-1}$, we add L lines of β processes. These new processes satisfy:
   - $\forall i \in \{1, \ldots, L\}, \forall j \in \{1, \ldots, \beta\}$:
     - $q_{(i,j)}.id = (i-1)\beta + j$
     - $q_{(i,j)}.leader = 0$
     - $q_{(i,j)}.level = i$

Algorithm 3 Algorithm of the daemon.

```
function Daemon
    for i = 1 ... β, (i++) do
        Reset(3,i);
        if i < β then
            Build(3,i+1);
        end if
    end for
end function

function Reset(layer, column)
    for i = top(layer + 1) ... bottom(layer + 1), (i++) do
        p_(i,column) executes R;
    end for
    if layer = 1 then
        for j = 1 ... β, (j++) do
            p_(L,j) executes R;                              ▷ Reset of layer 1
        end for
    else
        for j = 1 ... β, (j++) do
            Reset(layer − 1, j);
            if j < β then
                Build(layer − 1, j + 1);
            end if
        end for
    end if
end function
```

Algorithm 4 Algorithm of the daemon.

```
function Build(layer, column)
    for i = top(layer) ... bottom(layer), (i++) do
        for j = 1 ... β, (j++) do
            p_(i,j) executes J;
        end for
        for k = i − 1 ... 2(i − L/2), (k−−) do
            if k ≥ top(layer) then
                for j = 1 ... β, (j++) do
                    p_(k,j) executes C1;
                end for
            else
                p_(k,column) executes C1;
            end if
        end for
        for k = i ... 2(i − L/2) + 1, (k−−) do
            if k ≥ top(layer) then
                for j = 1 ... β, (j++) do
                    p_(k,j) executes C2;
                end for
            else
                p_(k,column) executes C2;
            end if
        end for
    end for
    if layer > 1 then
        Build(layer − 1, 1);
    end if
end function
```

– $q_{(i,j)}.done =$ `false`

- $\forall i \in \{2, \ldots, L\}, \forall j \in \{1, \ldots, \beta\}$, $\{q_{(i-1,j)}, q_{(i,j)}\} \in E$ and $q_{(i,j)}.parent = q_{(i-1,j)}$
- $\forall j \in \{1, \ldots, \beta\}, q_{(1,j)}.parent = q_{(1,j)}$
- $\forall j \in \{2, \ldots, \beta\}, \forall i \in \{1, \ldots, L\}$, $q_{(i,j)}.color = 2$
- $\forall i \in \{1, \ldots, L\}$, $q_{(i,1)}.color = (i \mod 2) + 1$

3. The former first line of $E_{\alpha-1}$ becomes a new bipartite complete subgraph with the last added line:

   - $\forall j \in \{1, \ldots, \beta\}, \forall j' \in \{1, \ldots, \beta\}$, $\{q_{(L,j)}, q_{(L+1,j')}\} \in E$
   - $\forall j \in \{1, \ldots, \beta\}$, $q_{(L+1,j)}.parent = q_{(L,1)}$

Figure 13 shows the structure of the network for E_5 and its initial configuration.

In the execution, the daemon selects processes according to function Daemon(α) (see Algorithm 5) which is the generalization of the algorithm presented in Section 5.2. In $E_{\alpha-1}$, processes $p_{(L,.)}$ execute $\beta^{\alpha-1}$ times Action R. Now, we added a new level of recursion. Then, processes $q_{(L',.)}$ execute β^{α} times Action R. As $\beta = \frac{n}{L'}$, the execution lasts $\Omega(n^{\alpha})$ steps. Hence, we obtain:

Theorem 13. *For every $\alpha \geq 4$, for every $\beta \geq 2$, there exists a network E_{α} of $n = 2^{\alpha-1} \times \beta$ processes in which there exists a possible execution of Algorithm $\mathcal{DLV}$ that stabilizes in $\Omega(n^{\alpha})$ steps.*

Algorithm 5 Generalization of the algorithm of the daemon for E_{α}.

function Daemon(α)
  for $i = 1 \ldots \beta$, $(i++)$ **do**
    Reset($\alpha - 1$,i); ▷ See Algorithm 3
    if $i < \beta$ **then**
      Build($\alpha - 1$,i+1); ▷ See Algorithm 4
    end if
  end for
end function

We proved that for E_{α} of size $n = L \times \beta$ ($\beta \geq 2$, $\alpha \geq 4$ and $L = 2^{\alpha-1}$), the execution in Algorithm 5 stabilizes using at least β^{α} steps. For a fixed size n of network, the value β^{α} may vary, depending on *e.g.* L. For instance, for $L = n/2$, we have that $\alpha = \log_2 n$ and $\beta = 2$ which implies that $\beta^{\alpha} = n$. At the opposite of the interval of L (second example), when $L = 8$, we have $\alpha = 4$ and $\beta = n \times 2^{-3}$. Hence, in this case, $\beta^{\alpha} = 2 \times n^4$. Both costs obtained in those examples are polynomial. But, between them, the function reaches higher values: the following corollary shows that the highest value of β^{α} is reached for $L = \sqrt{\frac{n}{2}}$ and is non-polynomial.

Corollary 5. *The stabilization time of Algorithm $\mathcal{DLV}$ is in $\Omega\left((2n)^{\frac{1}{4}\log_2(2n)}\right)$ steps.*

Proof. We show below that for every $\alpha \geq 4$, for every $\beta \geq 2$, there exists a network of size $n = 2^{\alpha-1} \times \beta$ for which there exists an execution which stabilizes in $\Omega\left((2n)^{\frac{1}{4}\log_2(2n)}\right)$ steps.

Let β, α and L be positive integers such that $n = L \times \beta$, $\beta \geq 2$, $\alpha \geq 4$, and $L = 2^{\alpha-1}$ (as for Theorem 13). The value of the function β^{α} reaches its maximum when $L = \sqrt{\frac{n}{2}}$, $\beta = \sqrt{2n}$ and $\alpha = \frac{1}{2}(\log_2 n + 1)$. (This can be easily proved by cancellation of the derivative of β^{α} *w.r.t.* L.) In this case, β^{α} equals $(2n)^{\frac{1}{4}\log_2(2n)}$, and we are done. $\square$

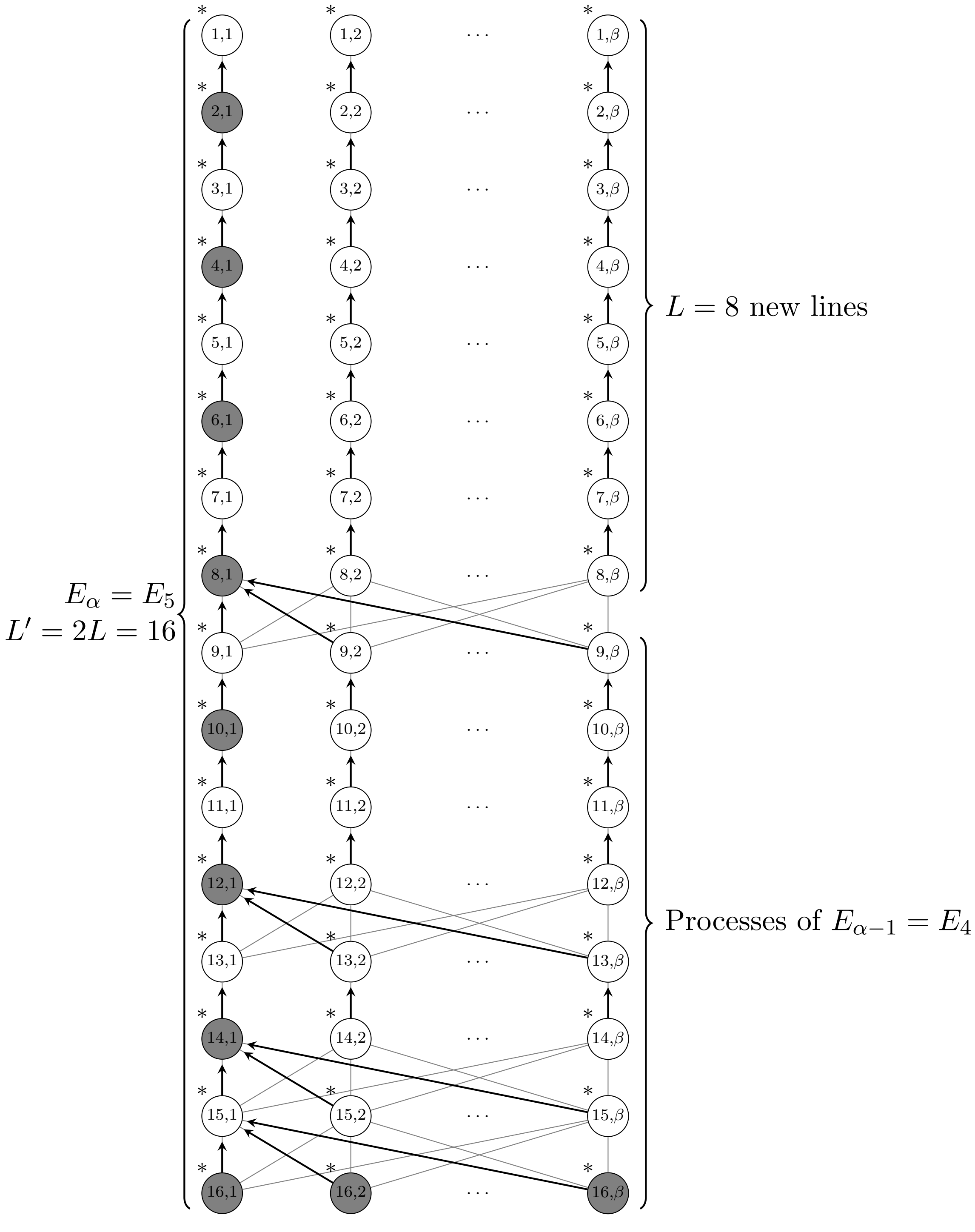


Figure 13: Initial configuration of the example in $\Omega(n^5)$ steps.

6 Step Complexity of Algorithm $\mathcal{DLV}2$

In this section, we study the step complexity of the algorithm given in [9], called here $\mathcal{DLV}2$.

6.1 Overview of $\mathcal{DLV}2$

The formal code of Algorithm $\mathcal{DLV}2$ is given in Algorithm 6.[3] The principle of Algorithm $\mathcal{DLV}2$ is very similar to Algorithm $\mathcal{DLV}$. It elects ℓ and builds a breadth-first search spanning tree rooted at ℓ. A variable *leader* is used to save the ID of the current leader. The level of the process in the tree is saved into variable *level*. The *key* of a process p is the combination of its two variables $p.leader$ and $p.level$. The keys are ordered using the lexical order. Notice that there is no explicit pointer to the parent but it can easily be computed with the keys.

Notice that we suppose every ID to be different than 0. When there is a smaller possible key in the neighborhood of a process p, p may execute Action A_2 and update its key accordingly.

As in $\mathcal{LE}$ and $\mathcal{DLV}2$, a "good relation" between a process p and its parent, called $Valid(p)$, is defined. This predicate ensures that p is either a self root ($\langle p, 0\rangle$), a zero root ($\langle 0, 0\rangle$), or its key is greater or equal to the best possible key.

Algorithm 6 Algorithm $\mathcal{DLV}2$ [9] for every process p

Variables

$p.leader \in \mathbb{N}$, $p.level \in \mathbb{N}$, $p.key = \langle p.leader, p.level\rangle$

Macros

$$\begin{array}{lcl}
successor(\langle lead, lvl\rangle) & \equiv & \langle lead, lvl+1\rangle \\
MinKeyMeighbor(p) & \equiv & \min\{q.key : q \in \mathcal{N}_p\}
\end{array}$$

Predicates

$$\begin{array}{lcl}
SelfRoot(p) & \equiv & p.key = \langle p, 0\rangle \\
ZeroRoot(p) & \equiv & p.key = \langle 0, 0\rangle \\
Valid(p) & \equiv & SelfRoot(p) \vee ZeroRoot(p) \vee (p.key > MinKeyNeighbor(p)) \\
Is_Linked(p) & \equiv & p.key = successor(MinKeyNeighbor(p)) \\
Is_Good(p) & \equiv & Is_Linked(p) \vee (SelfRoot(p) \Rightarrow p.key < MinKeyNeighbor(p)) \\
 & & \quad \vee ZeroRoot(p) \\
Frozen(p) & \equiv & SelfRoot(p) \wedge (\exists q \in \mathcal{N}_p : q.leader = 0) \\
ZeroLeaf(p) & \equiv & (p.leader = 0) \wedge (\forall q \in \mathcal{N}_p : (q.key \leq p.key) \vee SelfRoot(q))
\end{array}$$

Actions

$$\begin{array}{llcll}
A1 & \text{(priority 1)} & :: & \neg Valid(p) & \rightarrow \ \textbf{if } p.leader < p.id \textbf{ then} \\
 & & & & \qquad p.key \leftarrow \langle 0, 0\rangle \\
 & & & & \quad\ \textbf{else} \\
 & & & & \qquad p.key \leftarrow \langle p, 0\rangle \\
A2 & \text{(priority 2)} & :: & \neg Is_Good(p) \wedge & \rightarrow \ p.key \leftarrow successor(MinKeyNeighbor(p)) \\
 & & & \neg Frozen(p) & \\
A3 & \text{(priority 3)} & :: & ZeroLeaf(p) & \rightarrow \ p.key \leftarrow \langle p, 0\rangle
\end{array}$$

Zero propagation The main difference between $\mathcal{DLV}$ and $\mathcal{DLV}2$ is the way to deal with fake IDs. $\mathcal{DLV}2$ exploits the value 0, smaller than any ID. More precisely, if a process p is not valid

[3] The code given in Algorithm 6 is slightly different from the one given in [9]. Actually, we found a flaw in the definition of the *Valid* predicate. After private communication with the authors, we agree on the solution proposed here.

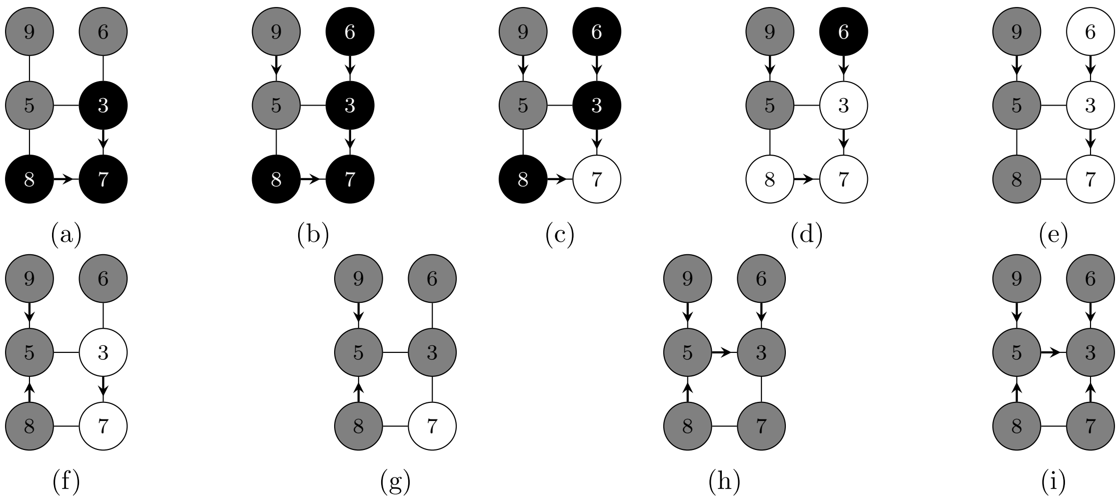


Figure 14: Example of execution of algorithm $\mathcal{DLV}2$.

and if its leader is smaller than its own ID, *i.e.* maybe a fake ID, p executes Action A_1 and gets key $\langle 0, 0\rangle$. 0 is then propagated in the network using Action A_2 and erase any fake ID. The only processes that can make a barrier to the propagation of 0 are the self roots. Indeed a self root neighbor with a process of leader 0 is said frozen, *i.e.* it cannot execute Action A_2 and get 0 as leader too.

When the growing of zero trees ends, the leaves, *i.e.* processes of leader 0 that are surrounded by self roots or processes with smaller key, can reset to self root executing Action A_3.

Example of execution Figure 14 shows an example of execution of $\mathcal{DLV}2$. For an easy reading of the figure, we explicit the parent pointers. The colors of processes are used to differentiate the leader. If the node is gray, its leader is an existing ID (we can infer which one with the parent pointers). If the node is black, its leader is the fake ID 1, smaller than any ID in the network. If the node is white, its leader is 0. We can also infer the level with the parent pointers.

In the initial configuration (Configuration a), the leader of processes 3, 7, and 8 is the fake ID 1. Then, in step a $\mapsto$ b, 1 is propagated to process 6 that executes A_2. At the same time, 9 also executes A_2 and chooses 5 as leader. 7 corrects its error and becomes a zero root by executing Action A_1 during step b $\mapsto$ c. The special ID 0 is propagated to processes 3 and 8 in step c $\mapsto$ d and then to process 6 in step d $\mapsto$ e. At the same time, 8 can reset itself executing Action A_3 because it is a zero leaf. Notice that 0 is not propagated to 5 since 5 is a self root and cannot execute Action A_2. In step e $\mapsto$ f, 6 resets itself and 8 executes Action A_2 to choose 5 as leader. Then, 3 executes Action A_3 during step f $\mapsto$ g. The last process of leader 0, process 7, resets itself during step g $\mapsto$ h. In the same step, 5 and 6 execute Action A_2 and choose 3 as leader. Notice that the leader of 8 and 9 is still 5 in Configuration h. Leader 3 is propagated to 7, 8, and 9 during step h $\mapsto$ i. Hence, 3 is elected in Configuration i.

6.2 Example of Exponential Execution

Network and initial configuration We consider a network composed of $n \geq 5$ processes p_k of ID $k \in \{2, \ldots, n+1\}$. Notice that 1 is a fake ID smaller than every ID in the network. Figure 15 shows the network and the initial configuration. The network is composed of $H = \left\lfloor \frac{n-1}{4} \right\rfloor$ *diamonds*. $\forall h \in \{0, \ldots, H-1\}$, *Diamond* h is made of the following edges: $\{p_{4(H-h-1)+2}, p_{4(H-h-1)+3}\}$, $\{p_{4(H-h-1)+2}, p_{4(H-h-1)+4}\}$, $\{p_{4(H-h-1)+3}, p_{4(H-h-1)+5}\}$, $\{p_{4(H-h-1)+4}, p_{4(H-h-1)}$

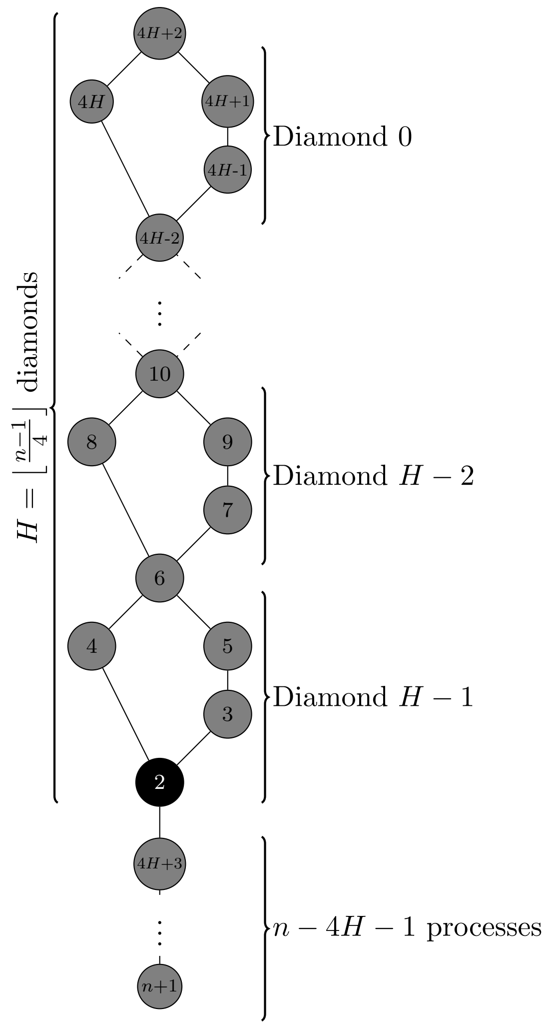


Figure 15: Initial configuration for any $n \geq 5$.

and $\{p_{4(H-h-1)+5}, p_{4(H-h-1)+6}\}$. The remaining processes form a chain linked to p_2, *i.e.* the edges $\{p_i, p_{i+1}\}$ with $i \in \{4H+4, n\}$, and the edge $\{p_2, p_{4H+3}\}$.

We consider the initial configuration where $p_2.key = \langle 1, 0 \rangle$, *i.e.* p_2 has the fake id 1 as leader, and $\forall i \in \{3, \ldots, n+1\}$, $p_i.key = \langle i, 0 \rangle$, *i.e.* p_i is self root.

Overview of the execution for $n = 11$ Figure 16 shows an intuitive idea of the execution for $n = 11$. Each phase is composed of three waves: the propagation of fake ID 1, the propagation of special ID 0, and the reset.

During the first phase ((1) in Figure 16), the fake ID 1 is propagated to p_4, p_6, p_8, and p_{10}. The fake ID 1 is also propagated to p_3 and p_7 to prepare the next phases. Then, p_2 corrects its error executing Action A_1 and the special ID 0 is propagated along the same path. The reset starts at p_{10}, and then p_8 resets.

p_7 still has 1 as leader so, in phase (2), 1 is propagated to p_9 and p_{10}. Then, since p_6 holds 0, 0 propagated to p_7, p_9, and p_{10}. Finally, p_{10}, p_9, p_7, p_6 and p_4 resets.

Then, during phase (3), we start again on the right side. $p_3.leader = 1$ so 1 is propagated to p_5, p_6, p_8, and p_{10}. Then 0 is propagated to p_3 and along the same path. The reset starts from p_{10} to p_8 as in phase (1).

Finally phase (4) is similar to phase (2) with a reset along the right side of the network.

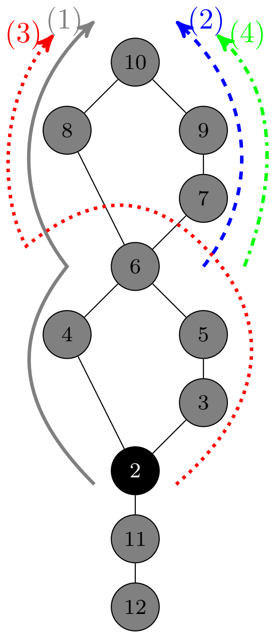


Figure 16: Intuitive idea of the execution for $n = 11$.

Notice that the additional processes p_{11} and p_{12} do nothing.

Generalization for any $n \geq 5$ We generalize this idea for any $n \geq 5$. We consider an unfair daemon that selects the enabled processes according to function DAEMON given in Algorithm 7.

Theorem 14. *For every $n \geq 5$, there exists a network of n processes in which there exists a possible execution of Algorithm $\mathcal{DLV}2$ that stabilizes in $\Omega(17 \times (2^{\lfloor \frac{n-1}{4} \rfloor} - 1)$ steps.*

Proof. Let consider the diamond h. When $p_{4(H-h-1)+2}$ holds 1 as leader, the processes into diamond h executes the following actions:

- Propagation of 1 on the left: $p_{4(H-h-1)+3}$, $p_{4(H-h-1)+4}$, $p_{4(H-h-1)+6}$ executes Action A_2
- Propagation of 0 on the left: $p_{4(H-h-1)+2}$ executes Action A_1, $p_{4(H-h-1)+4}$, $p_{4(H-h-1)+6}$ executes Action A_2
- Reset on the left: $p_{4(H-h-1)+6}$, $p_{4(H-h-1)+4}$ executes Action A_3
- Propagation of 1 on the right: $p_{4(H-h-1)+5}$, $p_{4(H-h-1)+6}$ executes Action A_2
- Propagation of 0 on the right: $p_{4(H-h-1)+3}$, $p_{4(H-h-1)+5}$, $p_{4(H-h-1)+6}$ executes Action A_2
- Reset on the right: $p_{4(H-h-1)+6}$, $p_{4(H-h-1)+5}$, $p_{4(H-h-1)+3}$, $p_{4(H-h-1)+2}$

So we have 17 actions. Notice that $p_{4(H-h-1)+6}$ holds 1 as leader twice. Hence, if $h \geq 1$, one such execution on diamond h implies two executions on diamond $h-1$. We denote $T(h)$ the maximum number of actions executed by processes on diamonds h to 0. So $T(h) \geq 17 + 2T(h-1)$, for $h \geq 1$. Notice that this execution on diamond 0 does not imply any other actions, so $T(0) \geq 17$.

We can trivially prove by induction that $T(h) \geq 17 \sum_{i=0}^{h} 2^i$. Hence,

$$T(H-1) \geq 17 \sum_{i=0}^{H-1} 2^i = 17(2^H - 1) = 17(2^{\lfloor \frac{n-1}{4} \rfloor} - 1)$$

□

Algorithm 7 Algorithm of the daemon.

```
function Daemon
    p_3 executes Action A_2;
    BuildLeft(H − 1,1);
    p_2 executes Action A_1;
    BuildLeft(H − 1,0);
    ResetLeft(H − 1);
end function

function BuildLeft(h, b)
    p_{4(H−h−1)+4} executes Action A_2;
    p_{4(H−h−1)+6} executes Action A_2;
    if h > 0 then
        if b = 1 then
            p_{4(H−h−1)+7} executes Action A_2;
        end if
        BuildLeft(h − 1, b);
    end if
end function

function BuildRight(h)
    if b ≠ 1 then
        p_{4(H−h−1)+3} executes Action A_2;
    end if
    p_{4(H−h−1)+5} executes Action A_2;
    p_{4(H−h−1)+6} executes Action A_2;
    if h > 0 then
        if b = 1 then
            p_{4(H−h−1)+7} executes Action A_2;
        end if
        BuildLeft(h − 1, b);
    end if
end function

function ResetLeft(h)
    if h = 0 then
        p_{4∗(H−1)+6} executes Action A_3;
    else
        ResetLeft(h − 1);
    end if
    p_{4(H−h−1)+4} executes Action A_3;
    BuildRight(h, 1);
    BuildRight(h, 0);
    ResetRight(h);
end function

function ResetRight(h)
    if h = 0 then
        p_{4∗(H−1)+6} executes Action A_3;
    else
        ResetLeft(h − 1);
    end if
    p_{4(H−h−1)+5} executes Action A_3;
    p_{4(H−h−1)+3} executes Action A_3;
    p_{4(H−h−1)+2} executes Action A_3;
end function
```

7 Experimentations

We ran simulations to empirically evaluate and compare the average stabilization times of our algorithm $\mathcal{LE}$, Algorithm $\mathcal{DLV}$, and Algorithm $\mathcal{DLV}2$ in terms of steps and rounds.

Probabilistic daemon We use an event-driven homemade simulator dedicated to the locally shared memory model. In this simulator, we use a probabilistic daemon. To enforce asynchronism (*i.e.*, to maximize interleavings between actions at different processes), we implemented the daemon as follows: at each step, the probability that an enabled process is selected by the daemon follows an exponential distribution on the number of consecutive steps where the node was enabled yet not selected.

UDG In our experiments, we consider a particular family of random graphs: the *Unit Disk Graphs (UDGs)* [16]. Such graphs are generated as follows. First, nodes are deployed on a square plane uniformly at random. Then, two nodes are neighbors if and only if their Euclidean distance is smaller than a pre-defined radius. UDGs are commonly used to model wireless sensor networks. Indeed, sensors are motionless and equipped with radio. In this case, the pre-defined radius represents the range of radio antenna.

Arbitrary initial configuration In each experiment, the initial configuration is randomly generated. We lead the random generation in such a way that with probability $\frac{1}{2}$, the initial value of each *idR* variable (resp. each *leader* variable in Algorithms $\mathcal{DLV}$ and $\mathcal{DLV}2$) is a fake id. Moreover, the way IDs and *idR* (resp. *leader* in Algorithms $\mathcal{DLV}$ and $\mathcal{DLV}2$) are set allows the existence of fake IDs lower than the lowest existing ID of the network.

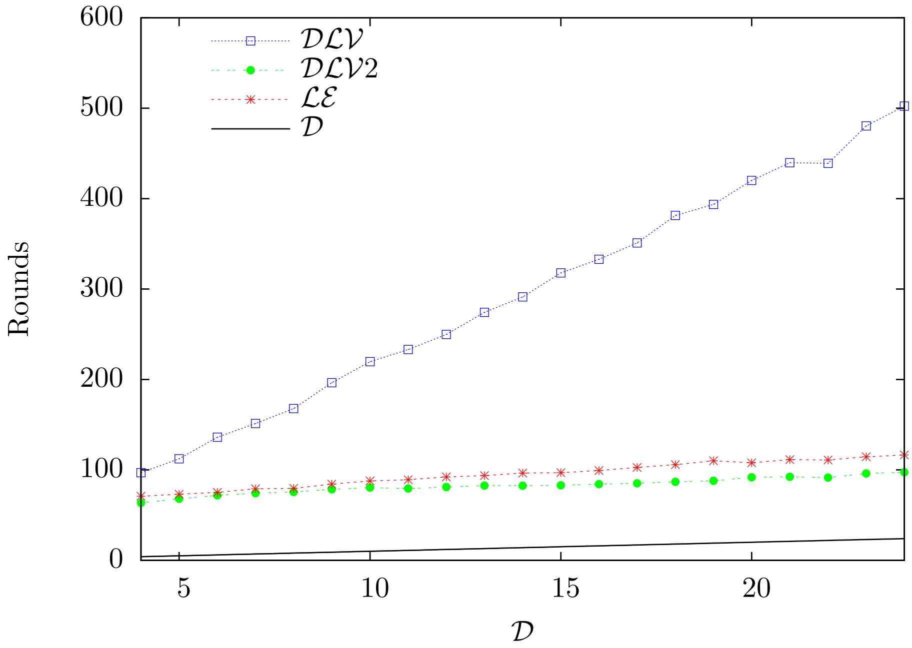


Figure 17: Average stabilization time in rounds for $\mathcal{LE}$, $\mathcal{DLV}$, and $\mathcal{DLV}2$ on UDGs ($n = 1000$).

Consider first Algorithms $\mathcal{LE}$ and $\mathcal{DLV}$. Each process selects its unique ID in $\{1, ..., 2n\}$ uniformly at random. Then, the initial value of each *idR* variable (resp. each *leader* variable in Algorithm $\mathcal{DLV}$) is chosen in $\{1, ..., 2n\}$ uniformly at random. The initial value of each *par* pointer is chosen among the ID of the node and the IDs of its neighbors uniformly at random. Finally, the initial value of each *level* variable is chosen in $\{0, ..., n-1\}$ uniformly at random. Finally, the initial value of all other variables (*i.e.*, *status* in $\mathcal{LE}$, and *color* and *done* in $\mathcal{DLV}$) are chosen uniformly at random in their respective (finite) domain.

The generation of the arbitrary initial configuration of $\mathcal{DLV}2$ is slightly different. Indeed, the *leader* variables can take the (reset) value 0, which is neither an ID, nor a fake ID. So, to ensure that the initial value of each *leader* variable is a fake id with probability $\frac{1}{2}$, (1) each process selects its unique ID in $\{1, ..., 2n+1\}$ uniformly at random, and (2) the initial value of each *leader* variable is chosen in $\{0, ..., 2n+1\}$ uniformly at random. Finally, the initial value of each *level* variable is chosen in $\{0, ..., n-1\}$ uniformly at random.

7.1 Average Stabilization Time in Rounds

In this subsection, we study the stabilization time in rounds. To that goal, we generated a pool of 220 UDGs. Each of those contains $n = 1000$ nodes. However, we make varying the diameters from 4 and 24 (20 graphs per diameter). We repeatedly executed each algorithm ($\mathcal{LE}$, $\mathcal{DLV}$, and $\mathcal{DLV}2$) on each graph with arbitrary initializations until obtaining a confidence interval smaller than 2% of the average stabilization time.

Results are shown on Figure 17. We can remark that the average stabilization time in rounds of $\mathcal{LE}$ is far from the analytical worst case (*i.e.*, $3n + \mathcal{D}$ rounds). Actually, we never observed an execution whose stabilization time matches the worst case. The average stabilization time in rounds of the three algorithms is sub-linear in n. However, in our experiments, our algorithm outperforms $\mathcal{DLV}$. Observed round complexities of $\mathcal{DLV}2$ are slightly better than those of our algorithm. This can be explained by the fact that $\mathcal{DLV}2$ removes abnormal trees in two waves, while our algorithm uses three waves. Finally, we draw the diameter $\mathcal{D}$ in the Figure to show

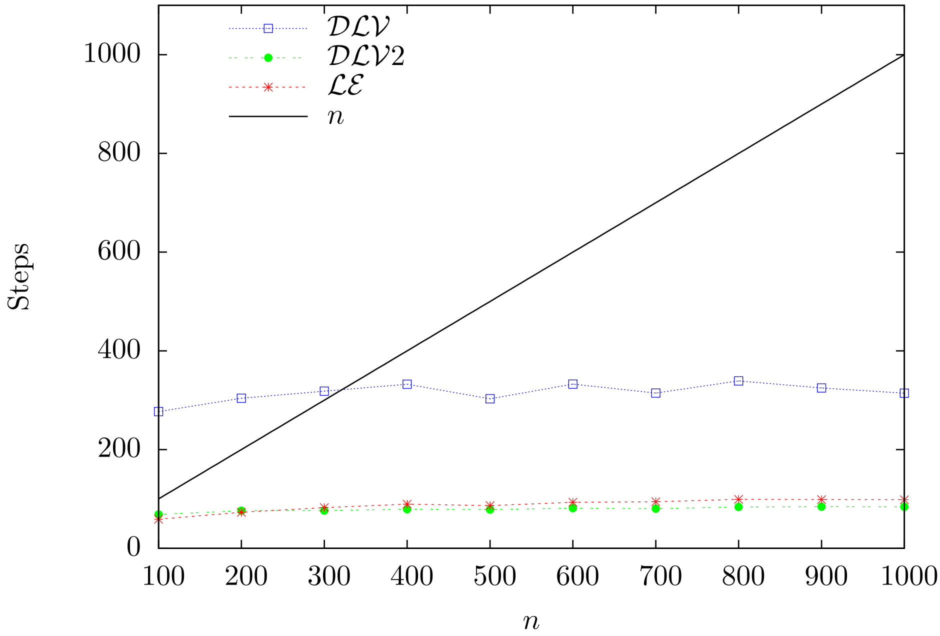


Figure 18: Average stabilization time in steps for $\mathcal{LE}$, $\mathcal{DLV}$, and $\mathcal{DLV}2$ on UDGs ($\mathcal{D} = 14$).

that the observed average stabilization times in rounds of $\mathcal{LE}$ and $\mathcal{DLV}2$ are of the order of magnitude of the diameter $\mathcal{D}$.

7.2 Average Stabilization Time in Steps

In this subsection, we study the average stabilization time in steps. To that goal, we generated a pool of 110 UDGs with diameter fixed to 14, yet where the number of nodes varies from 100 to 1000 (10 graphs per size). We repeatedly executed each algorithm ($\mathcal{LE}$, $\mathcal{DLV}$, and $\mathcal{DLV}2$) on each graph with arbitrary initializations until obtaining a confidence interval smaller than 2% of the average stabilization time in steps.

Results are shown on Figure 18. Again, we can remark that the average stabilization time in steps of $\mathcal{LE}$ is far from the analytical worst case (*i.e.*, $\Theta(n^3)$ steps). Again, we never observed an execution whose stabilization time matches the worst case. Again, in our experiments, $\mathcal{LE}$ outperforms $\mathcal{DLV}$, even though their respective average stabilization times in steps are of the same order of magnitude. Results observed with $\mathcal{LE}$ and $\mathcal{DLV}2$ are of the same order of magnitude. Finally, we draw the size n of the network in the Figure to show that the observed average stabilization times in steps of the three solutions do not depend on n.

8 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 $n \geq 4$, for every $\mathcal{D}, 2 \leq \mathcal{D} \leq n - 2$, there is a network of n processes in 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 $n \geq 4$, that there exists a network of n processes in which a possible execution exactly lasts $\frac{n^3}{6} + \frac{3}{2}n^2 - \frac{8}{3}n + 2$ steps, establishing then that the worst case is in $\Theta(n^3)$.

For fair comparison, we studied the step complexity of the previous best algorithms with similar settings (*i.e.*, they do not use any global knowledge and are proven assuming an unfair daemon) given in [10, 9] and respectively called here $\mathcal{DLV}$ and $\mathcal{DLV}2$. We showed that for a given $\alpha \geq 3$, for every $\beta \geq 2$, there exists a network of $n = 2^\alpha \times \beta$ processes in which there is an execution of algorithm $\mathcal{DLV}$ that stabilizes in $\Omega(n^{\alpha+1})$. In other words, the stabilization time of $\mathcal{DLV}$ in steps is not polynomial. Similarly, we showed that for any $n \geq 5$, there exists a network in which there is an execution of algorithm $\mathcal{DLV}2$ that stabilizes in $\Omega(2^{\lfloor \frac{n-1}{4} \rfloor})$ steps. Hence, the stabilization time of $\mathcal{DLV}2$ is also not polynomial.

Perspectives of this work deal with complexity issues. In [10], 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 $\mathcal{D}$). Nevertheless, processes are assumed to *know* D. It is worth investigating whether 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.

[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 *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 652–661, 1993.

[4] Lélia Blin, Alain Cournier, and Vincent Villain. An improved snap-stabilizing PIF algorithm. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003*, volume 2704 of *Lecture Notes in Computer Science*, pages 199–214, San Francisco, CA, USA, June 24-25 2003. Springer.

[5] Lélia Blin and Sébastien Tixeuil. Brief Announcement: Deterministic Self-stabilizing Leader Election with O(log log n)-bits. In *PODC*, pages 125–127, 2013.

[6] Janna Burman and Shay Kutten. Time Optimal Asynchronous Self-stabilizing Spanning Tree. In *Distributed Computing, 21st International Symposium (DISC)*, pages 92–107, 2007.

[7] Ernest J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.

[8] Ajoy Kumar Datta, Lawrence L. Larmore, and Hema Piniganti. Self-stabilizing Leader Election in Dynamic Networks. In *Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium (SSS)*, pages 35–49, 2010.

[9] 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.

[10] 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.

[11] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[12] Shlomi Dolev. *Self-stabilization.* MIT Press, March 2000.

[13] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory Requirements for Silent Stabilization. *Acta Inf.*, 36(6):447–462, 1999.

[14] Shlomi Dolev and Ted Herman. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[15] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform Dynamic Self-Stabilizing Leader Election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997.

[16] Mark L Huson and Arunabha Sen. Broadcast Scheduling Algorithms for Radio Networks. In *Military Communications Conference, 1995. MILCOM'95, Conference Record, IEEE*, volume 2, pages 647–651, 1995.

[17] Alex Kravchik and Shay Kutten. Time Optimal Synchronous Self Stabilizing Spanning Tree. In *DISC*, pages 91–105, 2013.

[18] Adrian Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.