Theoretical Computer Science 412 (2011) 5541–5561

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Self-stabilizing leader election in optimal space under an arbitrary scheduler[☆]

Ajoy K. Datta [*], Lawrence L. Larmore, Priyanka Vemula

School of Computer Science, University of Nevada, Las Vegas, United States

ARTICLE INFO

Keywords:
Distributed algorithm
Leader election
Self-stabilization
Silent algorithm
Unfair daemon

ABSTRACT

A silent self-stabilizing asynchronous distributed algorithm, SSLE, is given for the leader election problem in a connected unoriented (bidirectional) network with unique IDs. SSLE also constructs a BFS tree on the network rooted at that leader. SSLE uses $O(\log n)$ space per process and stabilizes in $O(n)$ rounds, against the unfair daemon, where n is the number of processes in the network.

1. Introduction

In distributed computing, *leader election* is the process of determining which individual process, in a network of processes, will be chosen by all processes to be the *leader*. Such an election could be a starting point for an algorithm which requires a single process to coordinate a task which requires cooperation among all processes.

In order for a leader to be elected in a general graph, symmetry must be broken; a standard method for breaking symmetry is to assume that each process has a unique ID. In this paper, we present a self-stabilizing and silent asynchronous leader election algorithm which elects the process of minimum ID to be the leader in any connected network, providing those processes have unique IDs, using the composite atomicity model of computation.

1.1. Related work

Arora and Gouda [2] give a silent leader election algorithm in the shared memory model. Their algorithm requires $O(N)$ rounds and $O(\log N)$ space, where N is a given upper bound on n, the size of the network. Dolev and Herman [7] give a non-silent leader election algorithm in the shared memory model. This algorithm takes $O(diam)$ rounds, where *diam* is the diameter of the network, and uses $O(N \log N)$ space. Awerbuch et al. [3] solve the leader election problem in the message passing model. Their algorithm takes $O(diam)$ rounds and uses $O(\log D \log N)$ space, where D is a given upper bound on the diameter.

Afek and Bremler [1] give an algorithm for the leader election problem in the message passing model. Their algorithm takes $O(n)$ rounds and uses $O(\log n)$ bits per process, where n is the size of the network. They do not claim that their algorithm works under the unfair daemon. Our algorithm SSLE is partially inspired by Afek and Bremler's algorithm.

1.2. Contributions

We give a uniform self-stabilizing distributed algorithm, SSLE, for the leader election problem in a connected network with unique IDs, using the composite atomicity model of computation. The proposed algorithm works under an arbitrary, *i.e.*,

[☆] An earlier version of this paper (Datta et al. [4]) was presented at SSS 2008.
[*] Corresponding author.
E-mail address: datta@cs.unlv.edu (A.K. Datta).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.05.001

unfair, scheduler (daemon). SSLE elects the process of least ID, which we call $\mathbb{L}$, to be the leader, and in addition, constructs a breadth first search (BFS) tree rooted at $\mathbb{L}$.

The space complexity of SSLE is $O(\log n)$ bits per process, where n is the size of the network, under the usual assumption that an ID is stored using $O(\log n)$ bits. This is asymptotically the minimum possible space, since a process needs to store its own ID. From an arbitrary initial configuration, SSLE elects the leader and builds the BFS tree rooted at the leader within $O(n)$ rounds, and is silent within $O(diam)$ additional rounds, where $diam$ is the diameter of the network. SSLE does not require knowledge of any upper bound on either n or $diam$.

1.3. Outline of the paper

We define our model of computation in Section 2. In Section 3, we introduce simplified algorithms, which solve the problem under relaxed specifications, or which take more time than necessary.

In Section 4, we informally discuss the problems that need to be solved to obtain a silent self-stabilizing algorithm with time complexity $O(n)$ rounds which works under the unfair daemon. In Section 5, we give a formal definition of SSLE.

In Section 6, we give an informal proof that SSLE is correct, satisfies the time complexity we have claimed, and is self-stabilizing and silent. The reader will fully understand SSLE by simply reading up to this section.

In Section 7, we give detailed formal proofs of the properties of SSLE. We conclude with Section 8.

2. Preliminaries

We assume that we are given a network of processes, each of which has a unique ID, which cannot be changed by our algorithm. Each process P has a set of *neighbors* $\mathcal{N}_P$. The neighbor relation is symmetric, *i.e.,* $P \in \mathcal{N}_Q$ if and only if $Q \in \mathcal{N}_P$. We also assume that the network is a connected graph; given any two processes P and Q, there is a *path* from P to Q, *i.e.,* a sequence of processes $P = P_0, P_1, \ldots, P_m = Q$ such that $P_{i-1} \in \mathcal{N}_{P_i}$ for all $0 < i \leq m$. We will write $||P, Q||$ for the length of the shortest path between P and Q. Then $diam = \min_{P,Q} ||P, Q||$.

A *self-stabilizing* [5,6] system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the system. In particular, a self-stabilizing algorithm distributed algorithm will eventually reach a *legitimate state* within finite time, regardless of its initial configuration, and will remain in a legitimate state forever. An algorithm is called *silent* if eventually all execution halts.

In the composite atomicity model of computation, each process has variables. Each process can read the values of its own and its neighbors' variables, but can write only to its own variables. We assume that each transition from a configuration to another, called a *step* of the algorithm, is driven by a *scheduler*, also called a *daemon*.

The *program* of each process consists of a finite set of *actions* of the following form: $<$ *label* $>::$ $<$ *guard* $> \longrightarrow <$ *statement* $>$. The *guard* of an action in the program of a process P is a Boolean expression involving the registers of P and its neighbors. The *statement* of an action of P updates one or more variables of P. An action can be executed only if it is *enabled*, *i.e.,* its guard evaluates to true. A process is said to be enabled if at least one of its actions is enabled. A *step* $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more enabled processes executing an action. Evaluation of all guards and execution of all statements of an action are presumed to take place in one atomic step. A distributed algorithm is called *uniform* if every process has the same program.

When a process P executes the statement of an action, there could be neighbors of P that are executing statements during the same step. We specify that P uses the current values of its own variables (which could have just been changed during the current step), but old values of its neighbors' variables, *i.e.,* values before the current step.

We use the *distributed daemon*. If one or more processes are enabled, the daemon *selects* at least one of these enabled processes to execute an action. We also assume that daemon is *unfair*, *i.e.,* that it need never select a given enabled process unless it becomes the only enabled process.

We define a *computation* to be a sequence of configurations $\gamma_p \mapsto \gamma_{p+1} \cdots \mapsto \gamma_q$ such that each $\gamma_i \mapsto \gamma_{i+1}$ is a step.

We measure the time complexity of SSLE in *rounds* [6]. The notion of *round* [6], captures the speed of the slowest process in an execution. We say that a finite computation $\varrho = \gamma_p \mapsto \gamma_{p+1} \mapsto \cdots \mapsto \gamma_q$ is a *round* if the following two conditions hold:

1. Every process P that is enabled at γ_p either executes or becomes neutralized during some step of ϱ. We say that a process P is *neutralized* at a step $\gamma \mapsto \gamma'$ if P is enabled at γ and not enabled at γ', but P does not execute during that step.
2. The computation $\gamma_p \mapsto \cdots \mapsto \gamma_{q-1}$ does not satisfy condition 1.

We call a computation of positive length which fails to satisfy condition 1 an *incomplete* round.

We define the *round complexity* of a computation to be the number of disjoint rounds in the computation. More formally, we say that a computation $\gamma_p \mapsto \cdots \mapsto \gamma_q$ has round complexity m if there exist indices $p = i_0 < i_1 < \cdots < i_{m-1} < q$ such that

1. $\gamma_{i_{j-1}} \mapsto \cdots \mapsto \gamma_{i_j}$ is a round for all $1 \leq j < m$,
2. $\gamma_{i_{m-1}} \mapsto \cdots \mapsto \gamma_q$ is either a round or an incomplete round.

Table 3.1
Algorithm I.

Label	Name	Guard		Statement
A1	Join(Q)	$Q \in \mathcal{N}_P$ $Q.key = Best\_Nbr\_Key(P)$ $Succ\_Key(Q) < P.key$	$\longrightarrow$	$P.key \leftarrow Succ\_Key(Q)$

We remark that an incomplete round could have infinite length, since the unfair daemon might never select an enabled process. But this cannot happen for the algorithm SSLE given in this paper. In Section 7.2, we show that every computation of SSLE is finite, *i.e.*, SSLE "works" under the unfair daemon.

3. Simple algorithms for leader election

Our leader election algorithm, SSLE, given in Section 5, may seem unnecessarily complex at first glance. In this section, we show how the leader election problem can be solved more simply, provided we relax the specification of the problem.

3.1. Algorithm I

We first present Algorithm I, which converges in $O(diam)$ rounds, but is not self-stabilizing. In Algorithm I, each process P has variables

- $P.id$, the ID of P, an unchangeable value of some ordered *ID type*. We will assume that an ID is a non-negative integer.
- $P.leader$, the *leader* of P, also of ID type. If $P.leader = Q.id$, then P has (perhaps only temporarily) selected Q to be the leader. As the algorithm progresses, $P.leader$ could change, but eventually $P.leader$ must equal $\mathbb{L}.id$ for all P.
- $P.level$, a positive integer. Eventually, $P.level$ must equal $||P, \mathbb{L}||$, the distance from P to $\mathbb{L}$.

Let $P.key = (P.leader, P.level)$, the *key* of P. We order keys lexically, *i.e.*, $P.key < Q.key$ if $P.leader < Q.leader$, or $P.leader = Q.leader$ and $P.level < Q.level$. We also define functions:

- $Best\_Nbr\_Key(P) = \min\{Q.key : Q \in \mathcal{N}_P\}$, the least (in the lexical ordering) key of any neighbor of P.
- $Succ\_Key(P) = (P.leader, P.level + 1)$, the *successor* of the key of P.

Throughout, we will let $Self\_Key(P) = (P.id, 0)$, and $Final\_Key(P) = (\mathbb{L}.id, ||P, \mathbb{L}||)$ for any process P. $Final\_Key(P)$ is the desired final value of $P.key$.

3.1.1. Action table syntax

Algorithm I is formally defined by its *action table*, Table 3.1, which consists of only one action, *Join*. If P executes *Join*(Q), then P "joins" the set of processes which have decided that $Q.leader$ is the leader.

Table 3.1 (as well as the other action tables in this paper) consists of four columns. The first column contains a formal *label* of each action, while the second column contains a descriptive name of the action. The third column, consisting of one or more *clauses*, contains the *guard* of the action. The guard of an action the conjunction of all its clauses. The fourth column contains the *statement* of the action. Each line is an instruction to change a variable of a process P; if there are multiple lines, they are to be executed in sequence.

If the initial configuration of the network is "clean," meaning that $P.key = Self\_Key(P) = (P.id, 0)$ for all P, then P will converge within *diam* rounds. Fig. 3.1 shows the last few steps of a computation of Algorithm I. Eventually, $P.key = Final\_Key(P)$ for all P.

Algorithm I is not self-stabilizing. If, in the initial configuration, $P.key \geq Final\_Key(P)$ for all P, and if $\mathbb{L}.key = (\mathbb{L}.id, 0)$, then Algorithm I will converge to the correct configuration; otherwise, it will not, because $P.key$ can never increase.

3.2. Resetting and Algorithm II

If $P.key < Final\_Key(P)$ for some P, then at least one of those processes will be able to detect that the network is not in a legitimate state. In Algorithm II, a process executes an action we call *Reset* in that case. Algorithm II has the same variables as Algorithm I, but has two more functions and one more action. The new functions are:

- $Has\_Super\_Key(P) \equiv P.key > Self\_Key(P)$, that is, P has a *super key*, a *key* which is larger than $Self\_Key(P)$. In this case, P knows that it must lower its *key*, since $Final\_Key(P) \leq Self\_Key(P)$.
- $False\_Loc\_Min(P) \equiv (P.key \neq Self\_Key(P)) \wedge (\forall Q \in \mathcal{N}_P : P.key \leq Q.key)$, P is a *false local minimum*. In this case, P cannot execute *Join*, and also knows its *key* is not correct.

Table 3.2 is the action table of Algorithm II.

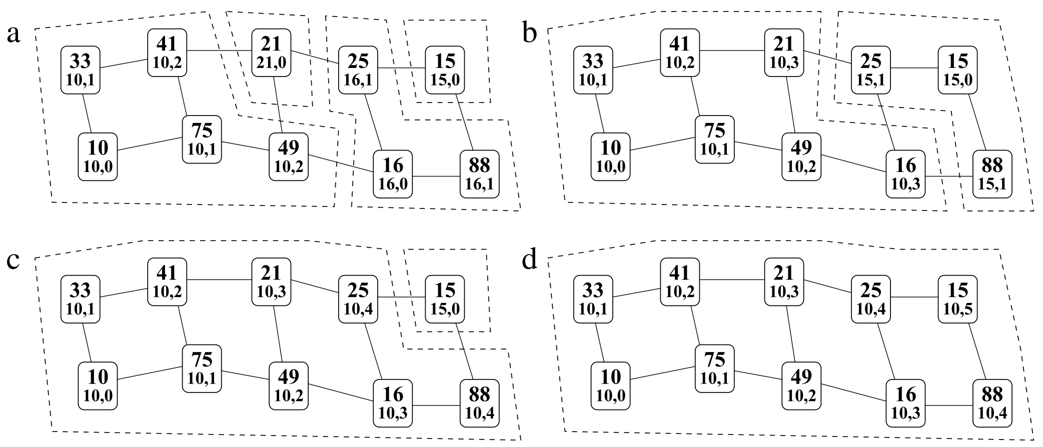


Fig. 3.1. An example of convergence of Algorithm I, where $\mathbb{L}.id = \mathbf{10}$. The ID of each process is shown as a larger numeral, while the *key* is shown as a pair of numerals below the ID. Each set of processes which acknowledge a given process to be the leader is shown within a dashed polygon. As an example of execution of *Join*, *Best_Nbr_Key*(**21**) = (**10**, 2) in configuration (a). In (b), after **21** executes, its *key* changes to (**10**, 3). Eventually, **10** is acknowledged to be the leader by all processes.

Table 3.2
Algorithm II.

Label	Name	Guard		Statement
B1	Join(Q)	$Q \in \mathcal{N}_P$ $Q.key = Best\_Nbr\_Key(P)$ $Succ\_Key(Q) < P.key$	$\longrightarrow$	$P.key \leftarrow Succ\_Key(Q)$
B2	Reset	$Has\_Super\_Key(P) \vee False\_Loc\_Min(P)$	$\longrightarrow$	$P.key \leftarrow Self\_Key(P)$

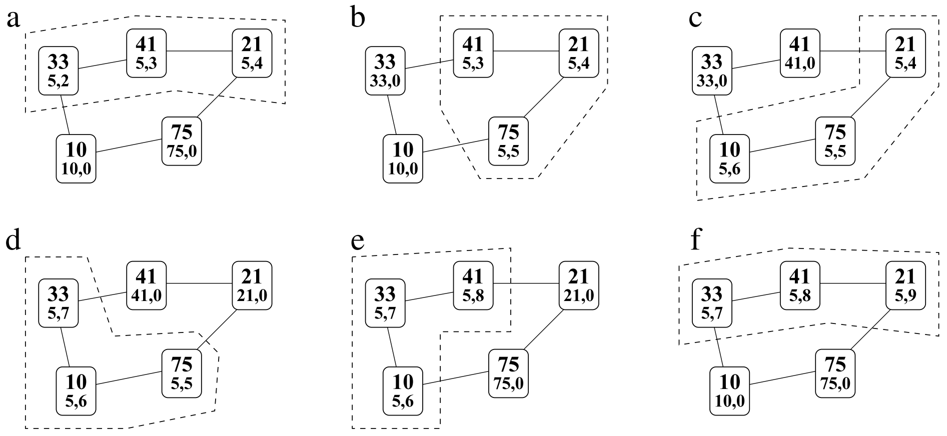


Fig. 3.2. The set of processes whose *leader* is the fictitious ID **5** is shown enclosed in a dashed polygon. Although each process in that set will eventually reset (*i.e.*, execute B2) additional processes can be recruited. The cycle of resetting and recruitment might never end, unless the value of *P.level* exceeds a pre-set limit.

3.3. The problem of fictitious leaders and Algorithm III

Algorithm II is not self-stabilizing, since *P.leader* could be initialized to a value of ID type which is not the ID of any process in the network, in which case we say that *P* has a *fictitious leader*. A fictitious leader that is greater than $\mathbb{L}.id$ is not a problem, but if a fictitious leader is less than $\mathbb{L}.id$, the network might never get rid of that fictitious ID. We illustrate this possibility in Fig. 3.2. Initially, in (a), the processes **33**, **41**, and **21** have *leader* = **5**. In each step shown, the false root executes Action B2, but another process chooses its *leader* to be **5**. If the values of *level* are allowed to increase without bound, the fictitious ID **5** might never be eliminated.

Using a known upper bound on the diameter. The problem of fictitious leaders can be solved if an upper bound, *D*, on the diameter of the network is given. We do not allow *P.level* to exceed *D*. Algorithm III, which includes this restriction as a clause of the *Join* action, then takes $O(D)$ rounds to converge.

Table 3.3
Algorithm III.

Label	Name	Guard		Statement
C1	Join(Q)	$Q \in \mathcal{N}_P$ $Q.key = Best\_Nbr\_Key(P)$ $Succ\_Key(Q) < P.key$ $Q.level < D$	$\longrightarrow$	$P.key \leftarrow Succ\_Key(Q)$
C2	Reset	$Has\_Super\_Key(P) \vee False\_Loc\_Min(P)$	$\longrightarrow$	$P.key \leftarrow Self\_Key(P)$

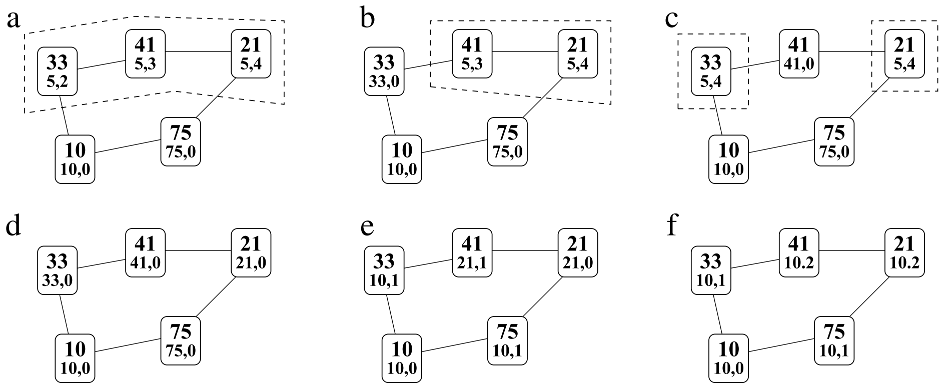


Fig. 3.3. Computation of Algorithm III, starting from Fig. 3.2(a). The set of processes whose *leader* is the fictitious ID **5** is shown enclosed in a dashed polygon. In this example, $D = 4$, and thus **75** is unable to execute *Join*(**21**). In (d), no processes have fictitious *leader* values. Within two rounds, Algorithm III converges to its final and correct configuration, shown in (f).

One problem with this approach is that we must, in advance, give a reasonable value of D. If D is less than *diam*, Algorithm III will not be correct, but if D is very much larger than *diam*, the algorithm takes unnecessarily long to converge (Table 3.3).

In Algorithm III, the growth of the set of processes following a fictitious leader must eventually halt, after which those nodes must eventually all reset, as we show in Fig. 3.3.

4. Informal overview of SSLE

In this section, we discuss the main ideas of SSLE, our algorithm for the leader election problem.

In SSLE, a process P has all the variables that are given for Algorithm III, plus $P.parent$, of ID type, $P.color \in \{1, 2\}$, and $P.done$, Boolean. We explain the purposes of each of these variables below.

4.1. Recruitment

As in Algorithms I, II, and III, a process P can choose a new key by joining with a neighbor Q; we say that P *attaches* to Q. When a process P attaches to a neighbor Q, it sets $P.parent$ to $Q.id$ and $P.key$ to $Succ\_Key(Q)$.

If the successor key of $P.parent$ is equal to $P.key$, then we say there is a *true parental link* from P to $P.parent$. The true parental links define a rooted spanning forest on the network, and the rooted trees of that spanning forest are called *components*.

4.2. False trees

If $P.parent = P$ and $P.key = Self\_Key(P)$, we call P a *true root*. If $P.parent = Q$ and $P.key = Succ\_Key(P)$, we say that P is a *true child of* Q. If P is neither a true root nor a true child, we call P a *false root*. All other processes are *true children*. Recall that a *component* is a maximal set of processes which is connected by true child pointers. Each component is rooted tree whose root is either a true root or a false root, and whose other members are true children. We refer to a component as a *true tree* or a *false tree*, respectively. All processes of a component have the same value of *leader*, which we call the *leader* ID of that component. We write *Component_Tree*(P) for the component which contains a given process P.

If a false tree consist of processes whose *leader* is greater than $\mathbb{L}.id$, that false tree will disappear naturally as a result of recruitment by processes whose *leader* values are smaller. A much more serious problem arises in the case that the leader ID of a component is less than $\mathbb{L}.id$. The processes of such a component cannot be recruited by *Component_Tree*($\mathbb{L}$), and that component could even recruit $\mathbb{L}$.

Our solution to this problem is for false roots to *reset*, in the same manner as in Algorithms II and III. If P detects that it is a false root, and it is unable to join a different component, it will change itself to a true root, by setting $P.parent$ to P and

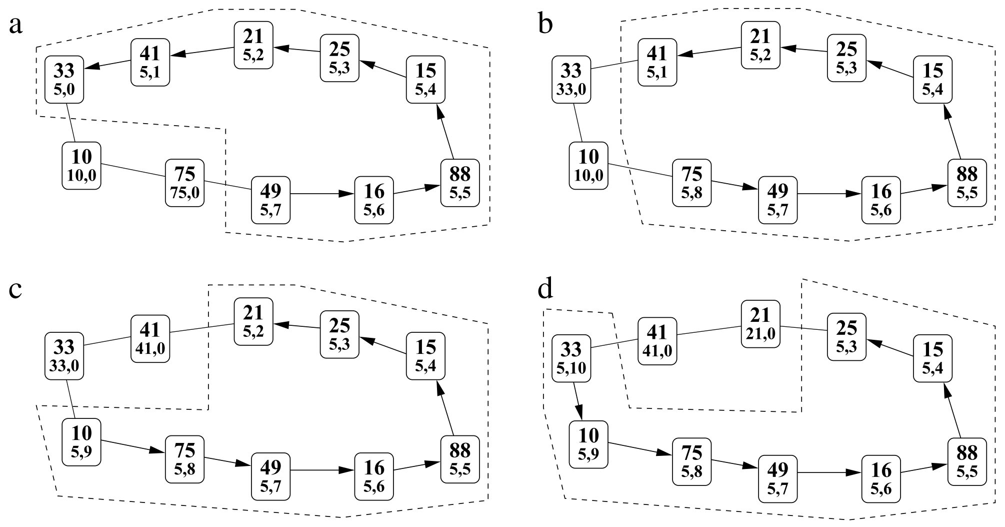


Fig. 4.1. Reset–recruitment cycle. The false tree shrinks at the root, but grows at the leaves.

P.key to *Self_Key*(*P*). When this happens, *P*'s erstwhile children will become false roots, and are enabled to reset, making their children false roots, and so forth. However, it is not clear that the false tree will ever disappear this way, since it could continually recruit processes at the leaves.

Fig. 4.1 shows a worst-case situation. In (a), there is a tree whose leader ID is the fictitious **5**. The root, **33** knows it is in error, and *resets*. At the next step, **41** realizes it is in error, and resets, and at the following step, **21** resets. However, meanwhile, the false tree is recruiting at the leaf end, and eventually recruits **33**.

This cycle can be stopped by choosing an upper bound on *P.level*, say *D*, as we did for Algorithm III. But picking a good value of *D* requires prior knowledge of the network. If *D* is large, the algorithm could take a long time to converge, while if *D* is less than the diameter of the network, the algorithm will not converge.

4.3. Color waves

We solve the problem of reset–recruitment cycles by introducing *colors*. Each process *P* will have a *color*, $P.color \in \{1, 2\}$. Using colors, we will introduce rules that prevent the indefinite growth of false trees:

1. Only a process of color 2 can recruit, and the recruited process must change its color to 1. That is, *P.parent* can be set to *Q* only if $Q.color = 2$, and if *P* makes that choice, then $P.color \leftarrow 1$.
2. If $P.color = 2$ and there are processes neighboring *P* that *P* believes it could recruit later, then *P.color* may not change to 1.
3. *P.color* can change only if *P* is a true root or a true child, $P.parent.color = P.color$, and all true children of *P* have the opposite color. (Note that, by vacuous implication, all children of a leaf have the opposite color.)

Colors change in convergecasting *color waves* initiated by the leaves and absorbed by a true root. False roots cannot absorb *color* waves, and *color* waves cannot pass each other. Thus, the *color* waves of a false tree must eventually stop moving. The rules that only processes of color 2 can recruit forces the creation of **two** *color* waves for each new layer added to a tree. Thus, a false tree eventually runs out of room to store new *color* waves.

Using the color rules listed above, we define a component to be *color locked* if no process in the component is enabled to change color, and no process is enabled join that component. For example, a false tree $\mathcal{C}$ is color locked if the following conditions hold:

- The colors of the processes of $\mathcal{C}$ alternate, *i.e.*, $P.parent.color \neq P.color$ if $P \in \mathcal{C}$ is a true child.
- No process can join $\mathcal{C}$, *i.e.*, if $Q \in \mathcal{N}_P$ for some $P \in \mathcal{C}$, then either $P.color = 1$ or $Q.level \leq Succ\_Key(P)$.

Fig. 4.2 shows how color locking limits the lifetime of a component with a fictitious *leader*.

4.4. Silence

After the final BFS tree is constructed, *color* waves could continue forever. We block this by introducing a Boolean variable *P.done* for every process *P*.

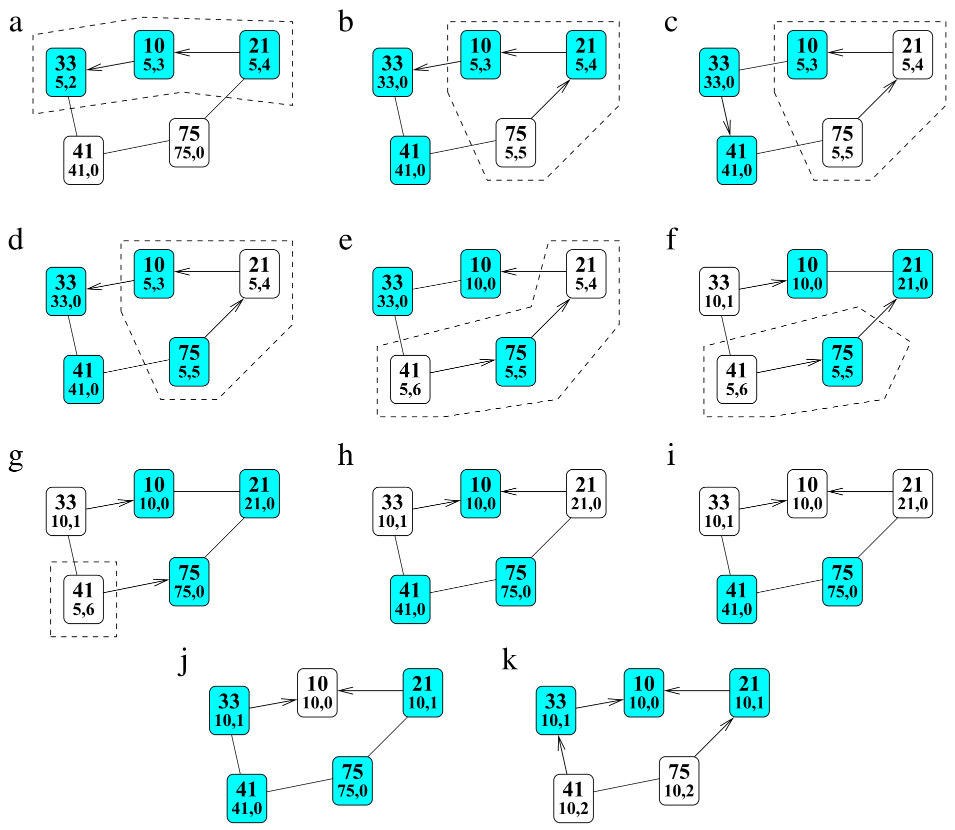


Fig. 4.2. A computation showing color locking, where $\mathbb{L} = \mathbf{10}$. In this and all subsequent figures in this paper, processes where *color* $= 2$ are shown in color, while processes where *color* $= 1$ are shown as colorless. The false tree whose *leader* is **5**, a fictitious ID, is shown enclosed by a dashed polygon at each step. At first, the false tree continues to recruit processes, even as resets from the root end. In the configuration shown at (e), the colors in the false tree prevent further recruitment. Eventually, the false tree disappears, permitting correct growth of the BFS tree, which is completed in (k). Not every step shown is a complete round; the round that starts at (b) takes four steps, since **10** and **33** are enabled at (b), but **10** is not selected until (d) and **33** is not selected until (e).

The following rules for setting the *done* variable are only informal. We give the exact formal rule in Section 5.

1. If a leaf P believes that the final BFS tree is finished, then $P.done$ is set to true.
2. If a process P with true children believes that the final BFS tree is finished, and $Q.done$ holds for all children of P, then $P.done$ is set to true.
3. If $P.done$ holds and P is a true root, then P cannot change its color.

We say that P is *color frozen* if $P.done$ and P is a true root.

If the root of a true tree $\mathcal{C}$ is color frozen, the colors of $\mathcal{C}$ alternate, and $\mathcal{C}$ is not able to recruit new members, then $\mathcal{C}$ is color locked.

Fig. 4.3 shows the sequence of configurations of a network after the final BFS tree has been constructed. A process P is shown as a double rounded rectangle if $P.done$ holds.

In (a), the leaves, **21**, **15**, and **88** set their *done* fields to TRUE. In the remaining figures up to (f), the *done* wave convergecasts to the root, **10**.

Starting from (f), **10** is color frozen, and hence unable to change color. Note that the *done* variable does not effect a process's ability to change color if it is not the root.

In (i), the colors of the tree alternate, and there are no other processes to recruit. The tree is color locked, and thus silence is achieved.

5. Formal description of SSLE

We now give the formal definition of SSLE. We first list the variables and the locally computable functions of SSLE, and then give the action table in Table 5.1. As before, we let $\mathbb{L}$ be the process of least ID in the network.

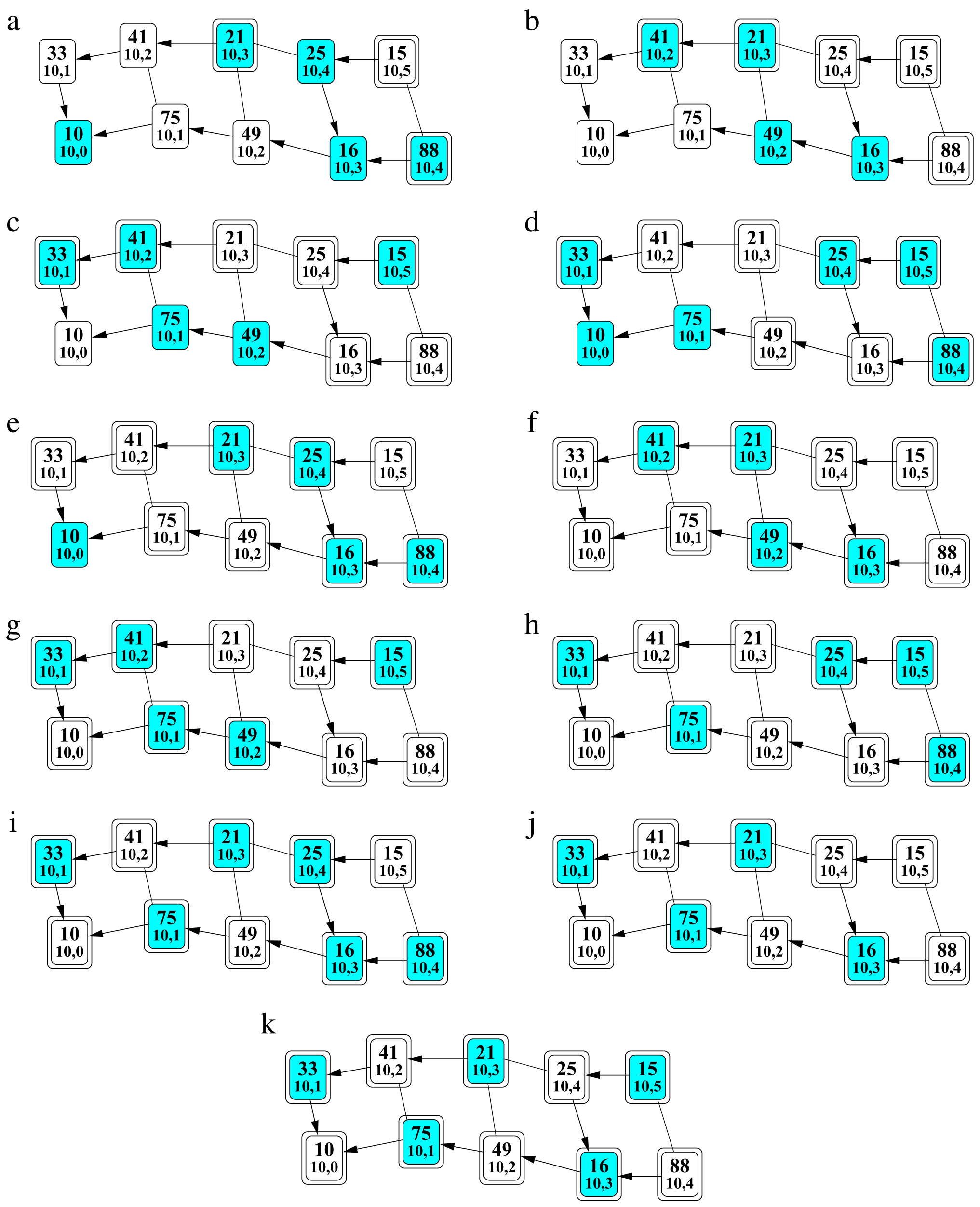


Fig. 4.3. Done wave convergecasts. If *P.done*, the diagram shows *P* enclosed in a double rounded rectangle. Starting at (f), **10** is color frozen, and thus can no longer change color. Other processes continue to change color until the tree is color locked at (k).

5.1. *Variables, functions and actions of* SSLE

Variables. Each process *P* has the following variables.

- *P.id*, the unique ID of *P*, of some ordered *ID type*, and cannot be altered by the algorithm.
  Without loss of generality, ID type is *unsigned integer*.
- *P.leader*, of ID type, *P*'s current estimate of the leader. $P.leader = \mathbb{L}$ eventually.
- *P.level*, a non-negative integer, *P*'s current estimate of its distance to the leader. Eventually, $P.level = ||P, \mathbb{L}||$.
- $P.key = (P.leader, P.level)$, the *key* of *P*. It is convenient to consider *P.key* as a variable of *P*, even though it is simply a combination of two variables and thus requires no extra space. Keys are ordered lexically; we say that $P.key < Q.key$ if $P.leader < Q.leader$, or $P.leader = Q.leader$ and $P.level < Q.level$.
- *P.parent*, of ID type. Except perhaps before *P* executes for the first time, *P.parent* will be the ID of either *P* or some neighbor of *P*, which we call the *parent* of *P*. When SSLE converges, the *parent* pointers will define a BFS tree rooted at $\mathbb{L}$.

Table 5.1
Actions of SSLE.

Label	Name	Guard		Statement
D1 priority 1	Join(Q)	$Is\_False\_Root(P) \vee (Succ\_Key(Q) < P.key)$ $Q.key = Best\_Nbr\_Key(P)$ $False\_Chldrn(P) = \emptyset$ $Q.color = 2$	$\longrightarrow$	$P.key \leftarrow Succ\_Key(Q)$ $P.parent \leftarrow Q$ $P.color \leftarrow 1$ $P.done \leftarrow$ FALSE
D2 priority 2	Reset	$Is\_False\_Root(P)$	$\longrightarrow$	$P.key \leftarrow Self\_Key(P)$ $P.parent \leftarrow P$ $P.color \leftarrow 2$ $P.done \leftarrow$ FALSE
D3 priority 3	Color 1	$P.color = 2$ $P.parent.color = 2$ $\forall Q \in True\_Chldrn(P) : Q.color = 1$ $\neg Color\_Frozen(P)$ $Recruits(P) = \emptyset$	$\longrightarrow$	$P.color \leftarrow 1$ $P.done \leftarrow Done(P)$
D4 priority 3	Color 2	$P.color = 1$ $P.parent.color = 1$ $\forall Q \in True\_Chldrn(P) : Q.color = 2$ $\neg Color\_Frozen(P)$	$\longrightarrow$	$P.color \leftarrow 2$ $P.done \leftarrow Done(P)$
D5 priority 4	Update Done	$P.done \not\equiv Done(P)$	$\longrightarrow$	$P.done \leftarrow Done(P)$

By an abuse of notation, we will identify the parent process of P with its ID whenever convenient. Thus, for example, if $P.parent = Q.id$, we will write $P.parent.key$ to denote $Q.key$.

- $P.color \in \{1, 2\}$, the *color* of P.
- $P.done$, Boolean. This predicate will be used to force SSLE to finish and become silent under the unfair daemon. When SSLE converges, $P.done$ will be true for all P.

Functions. The definition of SSLE makes use of a number of functions that can be computed by a process P.

- $Self\_Key(P) = (P.id, 0)$.
- $Is\_True\_Root(P) \equiv P.key = Self\_Key(P)$, P is a *true root*.
- $Succ\_Key(P) = (P.leader, P.level + 1)$, the *successor key* of P.
- $Is\_True\_Chld(P) \equiv (P.key = Succ\_Key(P.parent)) \wedge (P.leader < P.id)$, P is a *true child*. When SSLE converges, all processes except $\mathbb{L}$ will be true children.
- $Is\_False\_Root(P) \equiv \neg Is\_True\_Root(P) \wedge \neg Is\_True\_Chld(P)$, P is a *false root*. Every process is either a true root, a true child, or a false root. When SSLE converges, there will be no false roots.
- $Best\_Nbr\_Key(P) = \min\{Q.key : (Q \in \mathcal{N}_P) \wedge (Succ\_Key(Q) < Self\_Key(P)) \wedge (Q.color = 2)\}$, the least key of any neighbor P is allowed to choose as its parent.
- $True\_Chldrn(P) = \{Q \in \mathcal{N}_P : (Q.parent = P) \wedge (Q.key = Succ\_Key(P))\}$, the *true children* of P.
- $False\_Chldrn(P) = \{Q \in \mathcal{N}_P : (Q.parent = P) \wedge (Q.key \neq Succ\_Key(P))\}$, the *false children* of P. Note that all false children are false roots.
- $Recruits(P) = \{Q \in \mathcal{N}_P : Q.key > Succ\_Key(P)\}$, the *possible recruits* of P. These are neighbors of P which could eventually attach to P.
- $Done(P) \equiv (Recruits(P) = \emptyset) \wedge (\forall Q \in True\_Chldrn(P) : Q.done)$. The value of $P.done$ is set equal to the computed function $Done(P)$.
- $Color\_Frozen(P) \equiv Is\_True\_Root(P) \wedge P.done$, P is *color frozen*. This predicate will eventually hold only at $\mathbb{L}$, and will ensure that SSLE is silent.

Actions of SSLE.Table 5.1 lists the actions of SSLE. The syntax of that table is the same as that given in Section 3.1.1. The *label* of each action is listed in the first column, along with its priority number, and the second column contains an descriptive name of the action. The *guard* of each action is the conjunction of the clauses listed in the third column together with a priority rule. In order for an action to be enabled, all clauses of its guard must be true.

Officially, the guard of an action is the conjunction of its clauses, together with the clause that no action whose priority number if smaller is enabled. That is, if Action D1 is enabled, then no other action is enabled; if Action D2 is enabled, then the color actions and Action D5 are not enabled; and if a color action is enabled, Action D5 is not enabled.

The *statement* of each action is the list of commands given in the fourth column, which are executed in sequence, although that execution is presumed to take zero time.

5.2. Informal explanation of the actions

We refer to Actions D1 and D2 as *structure actions*, since they change the spanning forest. We refer to Actions D3 and D4 as *color actions*.

Action D1 causes a process P to join the component containing a neighbor Q. P can only execute this action if it will cause $P.key$ to decrease, and also only if $Q.color = 2$. The purpose of the color condition on Q is to slow the growth of its component; this is necessary to prevent false trees from recruiting new processes forever.

In order for P to execute D1, it may not have any false children. The purpose of this condition is to ensure that when a process joins a component, it does not cause additional processes to join as a side effect.

Action D2 causes a false root to *reset*, meaning that it changes to a true root. Its priority of 2 guarantees that P will not reset if it can join.

Action D3 and D4 cause P to advance a *color* wave or, in the case P is a leaf, to start a new *color* wave. If P is a true root when it changes color, the oldest *color* wave of its component is absorbed, *i.e.,* deleted. A false root cannot change color. Thus, a false root cannot absorb *color* waves, eventually causing a condition we call *color locking*, which bounds the growth of false trees.

If P is a true root and $P.done$, then P is not allowed to change color. This rule guarantees that SSLE is correct under the unfair daemon, and also that SSLE is silent.

The value of $P.done$ is corrected, *i.e.,* set to the function $Done(P)$, whenever P executes a color action. If P is unable to execute any structure action or color action, but $P.done$ is incorrect, it is enabled to correct the value of $P.done$ by executing Action D5.

6. Informal proof of SSLE

In this section, we give a complete, albeit informal, proof of the correctness and time complexity of SSLE. In Section 7, we give the proof in detail. The casual reader will be able to understand SSLE without reading Section 7.

We use the *convergence stair* method [8] to prove correctness and round complexity of SSLE. We first define a sequence of *benchmarks*, and we then prove that each benchmark is closed, and that once a given benchmark holds, the next benchmark will eventually hold.

6.1. Benchmarks

The motivation for some of our benchmarks is clear, but for others it may be a bit obscure. We will give an informal justification for each as we go. Recall that $\mathbb{L}$ denotes the process of least ID, and that the goal of SSLE is to elect $\mathbb{L}$ to be the leader.

Final processes. Eventually, $P.key$ must equal $Final\_Key(P)$. If $P.key = Final\_Key(P)$ and $P \in Component\_Tree(\mathbb{L})$, we say that P is *final*. SSLE is in a legitimate state if and only if all processes are final. Let *Final* be the set of all final processes.

Subfinal processes and pseudo-final processes. We define a process P to be *subfinal* if $P.key < Final\_Key(P)$. We define a process P to be *pseudo-final* if $P.key = Final\_Key(P)$, but $P \notin Component\_Tree(\mathbb{L})$.

Active processes. We define a process P to be *active* if it is final, and if there is some descendant of P in $Component\_Tree(\mathbb{L})$ which has a neighbor which is not in *Final*. More formally, we define *Active*, the set of *active* processes, recursively as follows:

- If $P \in Final$ and $\mathcal{N}_P \cap Final \neq \emptyset$, then $P \in Active$.
- If $P \in Active$ then $P.parent \in Active$.

Fig. 6.1 illustrates these sets in a simple example. Informally stated, a final process is active if it has a descendant which might still play a recruitment role.

Benchmark BM1: $P.key \leq Self\_Key(P)$ for all P.

Benchmark BM2: BM1 holds, and $P.key \geq Final\_Key(P)$ for all P. That is, there are no subfinal processes.

Benchmark BM3: BM2 holds, and $P.key = Final\_Key(P) \implies P \in Final$, that is, there are no pseudo-final processes.

Benchmark BM4: Benchmark BM3 holds, and $\neg P.done$ for all $P \in Active$. This condition ensures that *color* waves will continue to be absorbed by $\mathbb{L}$ as long as SSLE has not reached a legitimate state.

Benchmark BM5: Benchmark BM4 holds and all processes are final. When BM5 holds, the configuration is legitimate, but *color* and *done* waves can still be convergecast.

Benchmark BM6: Benchmark BM5 holds, and $P.done$ for all P. When BM6 holds, $\mathbb{L}$ is *color frozen*, and can no longer absorb *color* waves. *Color deadlock* will eventually result, as the *color* waves (which cannot pass each other) "pile up" like vehicles at an unmanned checkpoint.

Benchmark BM7: Benchmark BM6 holds, and, for all $P \neq \mathbb{L}$, $P.color \neq P.parent.color$. When BM7 holds, the network is in a legitimate state and is *color deadlocked*. $\mathbb{L}$ is color frozen, and for every $P \neq \mathbb{L}$, $P.parent.color \neq P.color$; thus no process can execute a color action. $P.done = Done(P)$ for all P, and thus no process can execute Action D5. Since all processes are in *Final*, no process can execute a structure action. Thus, SSLE is silent.

We now give informal proofs for each of the benchmarks.

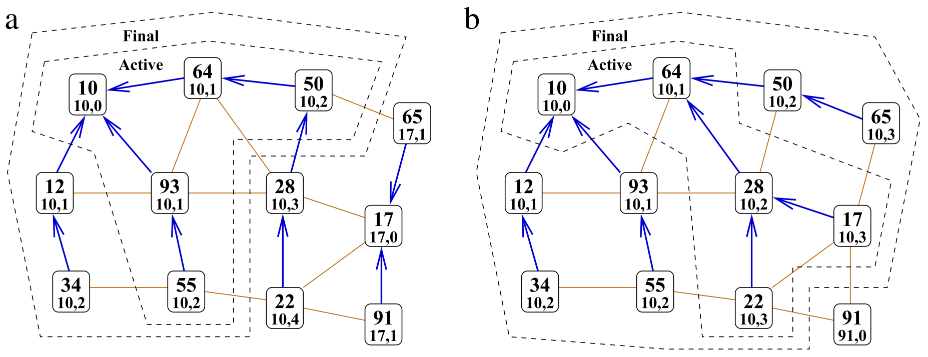


Fig. 6.1. Sets *Final* and *Active*. $\mathbb{L}.id = \mathbf{10}$. In (a), **28** is in *Component_Tree*($\mathbb{L}$) but is not final since its *level* is 3 and its distance to $\mathbb{L}$ is only 2. Similarly, **22** is not final. Of the final processes, only **12** and **34** are not active, since all other processes have descendants which are adjacent to possible future recruits. In (b), a number of steps later, **28** has corrected its *level*, and is now final. Both **17** and **22** have **91** as a possible future recruit, and thus are active, along with their ancestors, **28**, **64**, and **10**. All other final processes are not active.

6.1.1. Benchmark BM1

It is obvious to every process P, even though it does not know the identity of $\mathbb{L}$, that $\mathbb{L}.id \le P.id$. If, because of arbitrary initialization, $P.key > Self\_Key(P)$, then P knows that the key is too large, and will reset the very first time it executes an action, which means it will reset during the first round. There is no action that can cause $P.key$ to exceed $Self\_Key(P)$, and thus this first benchmark will hold if at least one round has been completed.

6.1.2. Benchmark BM2

SSLE uses *color locking* in two ways: first, to guarantee the elimination of false trees, and second, to guarantee silence.

For any given ℓ and any component $\mathcal{C}$, we refer to $\{P \in \mathcal{C} : P.level = \ell\}$ as a *layer* of $\mathcal{C}$. We can formally prove the color locking of a false tree by using the *energy* of a component, which roughly measures the ability of a component to grow. If P is a leaf of a component $\mathcal{C}$, let $Path(P)$ be the set of all processes on the path (following parent pointers) from P to the root of $\mathcal{C}$, let ℓ be the number of layers of $Path(P)$ and w the number of *color* waves of $Path(P)$. The energy of $Path(P)$ is $2\ell - w$ if $P.color = 1$, and $2\ell - w + 1$ if $P.color = 2$; the energy of $\mathcal{C}$ is defined to be the maximum of the energies of the paths from its leaves to its root. Note that the energy decreases every time the root of the false tree resets, and that the energy cannot be less than the number of layers of the false tree. Thus, energy of any false tree is bounded and must decrease every round. When the energy of a false component is equal to its number of layers, the component cannot add a new layer. Eventually, every false tree disappears.

We remark here on the connection with Afek and Bremler's algorithm [1]. In that algorithm, the concept of *power* is introduced. Essentially, the root of a true tree continually radiates *power*, permitting its tree to grow, while a false tree must run out of power and stop growing. We refer the reader to that paper for details.

In Fig. 6.2, we show a sequence of configurations chosen by an adversary who is trying to prolong the life of a false tree as much as possible. The false component shown in (a), rooted at the false root **33**, contains eight layers and five *color* waves. In (b), one layer has been lost, since **33** reset, making **41** the new false root, and one new layer was added by the recruitment of **75**. Each additional layer beyond that will require adding two more *color* waves, but the loss of a layer, caused by resetting of a false root, causes the loss of at most one *color* wave. The energy inexorably decreases, although not during every round. For example, a new *color* wave is added in (d), and another in (e), while the false tree acquires one new layer by recruiting **10**. Since a layer was lost by the resetting of **41**, the tree now has six *color* waves. Eventually, in (k), the false tree has nine layers and nine *color* waves. No new layers can be added, and the false tree will delete itself over the next nine rounds.

In (a), the energy of the false tree is 12. Energy cannot increase. In (b), the false tree has energy 11, in (c) and (d) the energy is still 11. In (e), (f), and (g), the energy is 10. In (h), (i), (j), and (k), it is 9, while in (l) it is 8.

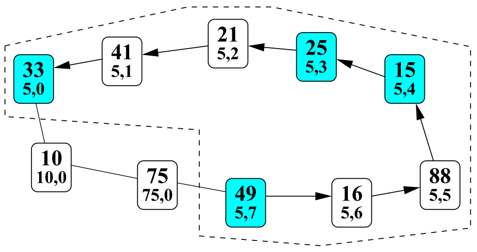
We give the complete formal proof that Benchmark BM2 will hold in $O(n)$ rounds in Section 7.1.2, where we give a recursive definition of energy.

6.1.3. Benchmark BM3

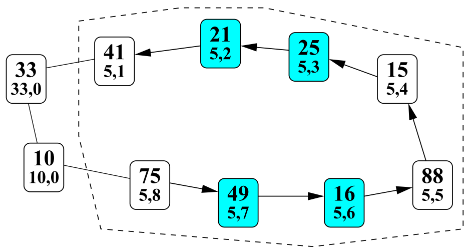
Suppose there are no subfinal processes. At each round, all pseudo-final processes of smallest *level* will reset, *i.e.,* execute Action D2, and thus the smallest *level* of any pseudo-final process will increase every round. Within at most *diam* additional rounds, there will be no pseudo-final processes, *i.e.,* Benchmark BM3 will hold.

6.1.4. Benchmark BM4

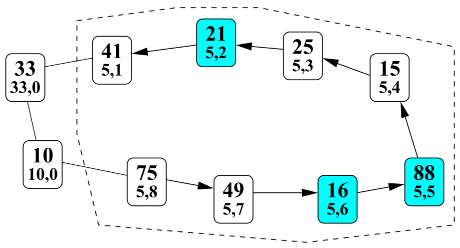
Suppose Benchmark BM3 holds. Since there are no subfinal processes, a final process P cannot execute D1, since there is no subfinal process for it to attach to, and it cannot execute D2, because either $P = \mathbb{L}$ or P is a true child. Thus, once P is final, it will remain final.



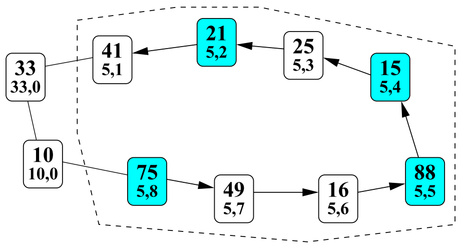
(a) $E_5 = 12$.



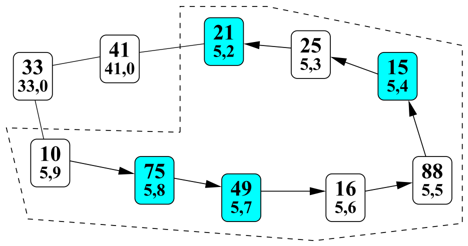
(b) $E_5 = 11$.



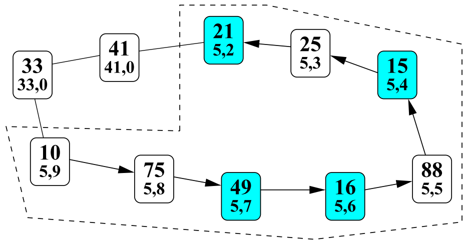
(c) $E_5 = 11$.



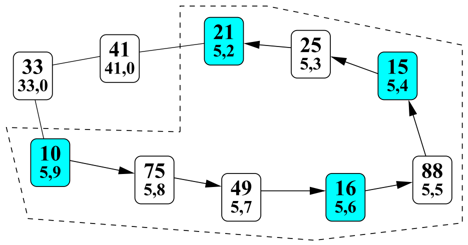
(d) $E_5 = 11$.



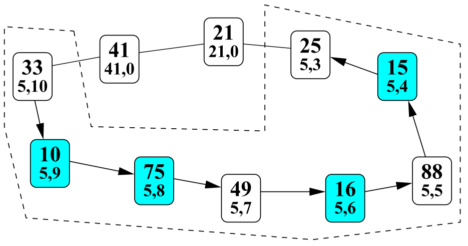
(e) $E_5 = 10$.



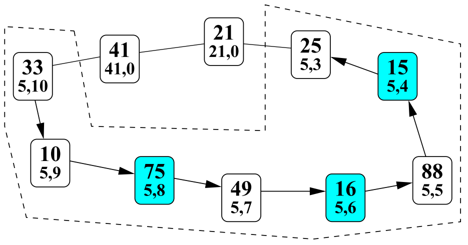
(f) $E_5 = 10$.



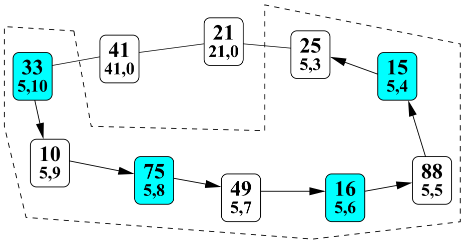
(g) $E_5 = 10$.



(h) $E_5 = 9$.



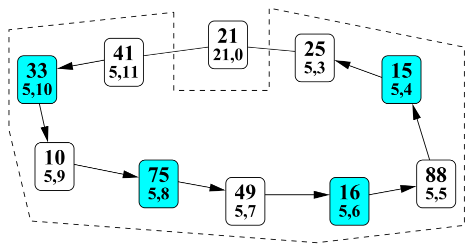
(i) $E_5 = 9$.



(j) $E_5 = 9$.

Fig. 6.2. Color locking bounds the growth of a false tree. In the computation shown, some rounds take more than one step. (a)–(b) is one round, (b)–(e) is one round, (e)–(h) is one round, and (h)–(l) is one round. The energy E of the false tree is shown in each figure, and decreases during every round. Once a false tree is color locked, as shown in (k), it remains color locked and can recruit no new processes, and shrinks every round. After $O(n)$ rounds, there are no false trees.

If P is an active leaf, then $Done(P)$ is false, which means that $P.done$ will be false within one round. If P is any active process, then, by induction $Done(P)$ will become false, followed by $P.done$. The number of rounds required for $P.done$ to become false does not exceed the distance from P to the nearest process which is not final, and that distance cannot exceed *diam*. Thus, Benchmark BM4 will hold within *diam* rounds.


(k) $E_5 = 9$.

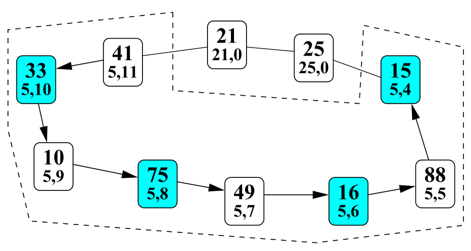

(l) $E_5 = 8$.

Fig. 6.2. *(continued.)*

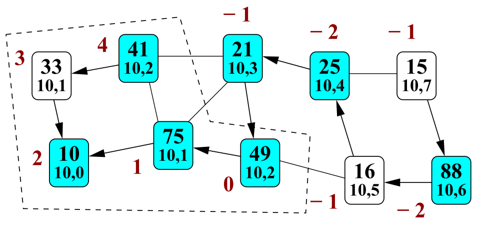


Fig. 6.3. Values of α are shown outside the processes. All processes are members of *Component_Tree*$(\mathbb{L})$. The set *Final* is enclosed in a dashed polygon.

6.1.5. Benchmark BM5

Assume Benchmark BM4 holds. We can show that Benchmark BM5 will hold within $O(n)$ rounds.

We define an integral function α on all processes in *Component_Tree*$(\mathbb{L})$. Let γ_0 be the first configuration at which Benchmark BM4 holds. We now define

- $\alpha(\mathbb{L}) = \begin{cases} 4\,(\text{the number of times } \mathbb{L} \text{ has executed D3 since } \gamma_0) & \text{if } \mathbb{L}.color = 1 \\ 4\,(\text{the number of times } \mathbb{L} \text{ has executed D3 since } \gamma_0) + 2 & \text{if } \mathbb{L}.color = 2 \end{cases}$
- $\alpha(P) = \begin{cases} \alpha(P.parent) - 1 & \text{if } P.color = P.parent.color \\ \alpha(P.parent) + 1 & \text{if } P.color \neq P.parent.color \end{cases}$

  if $P \in$ *Component_Tree*$(\mathbb{L})$, $P \neq \mathbb{L}$.
- $\alpha^{\min} = \min\{\alpha(P) : P \in \textit{Component\_Tree}(\mathbb{L})\}$.

We show an example of the function α in Fig. 6.3. In this figure, we assume that $\mathbb{L} = \mathbf{10}$ has not executed a color action since γ_0. In the example, *Component_Tree*$(\mathbb{L})$ contains all processes, and the members of *Final* are enclosed in a dashed polygon.

Initially, $-n+1 \le \alpha(P) \le n-1$. If $\alpha(P) = \alpha^{\min}$, then P is enabled to execute an action, and $\alpha^{\min}$ will increase during the next round. More specifically, if $\alpha(P) = \alpha^{\min}$, then P will not be enabled to execute Action D1, since if it were, the process it would attach to would have a smaller value of α. P will be enabled to execute a color action, which will increase $\alpha(P)$ by 2. We can show that $\alpha^{\min}$ must increase during every round as long as Benchmark BM5 does not hold. We can also show that, once $\alpha^{\min} \ge 3\,diam + 2$, Benchmark BM5 must hold. Thus, we reach the benchmark within $O(n)$ rounds.

We now construct an example that demonstrates that the number of rounds required to reach Benchmark BM5 is $\Omega(n)$ in the worst case. Consider the following network and initial configuration:

- There are n processes and $2n-3$ edges in the network. All other processes are neighbors of $\mathbb{L}$, and if $\mathbb{L}$ is removed, the remaining network is a chain. Thus, $diam = 2$.
- *Component_Tree*$(\mathbb{L})$ is a chain consisting of all processes.
- $P.color = 1$ for all P. Thus, $\alpha(P) = -||P, \mathbb{L}||$ for all P, and $\alpha^{\min} = -n+1$.

If every enabled process is selected at each round (so that each round consists of a single step) then increases by 1 every round. BM5 will be achieved within $n+1$ rounds, and $\alpha^{\min} = 3$ at that configuration.

Fig. 6.4 illustrates a computation of SSLE on that network, where $n = 7$, where the initial configuration is shown in (a). The values of α are shown adjacent to every process. We assume that at each step, every enabled process is selected, so each round consists of exactly one step. Benchmark BM5 holds for configuration shown in (i), which is reached after eight steps.

6.1.6. Benchmark BM6

Assume Benchmark BM5 holds. No further structure actions will be executed, and thus every time a process P executes an action, $P.done$ is set to $Done(P)$. Starting at the leaves of *Component_Tree*$(\mathbb{L})$, the *done* wave convergecasts to $\mathbb{L}$, after which $P.done$ will be true for all P. Thus, within $diam + 1$ rounds, Benchmark BM6 holds.

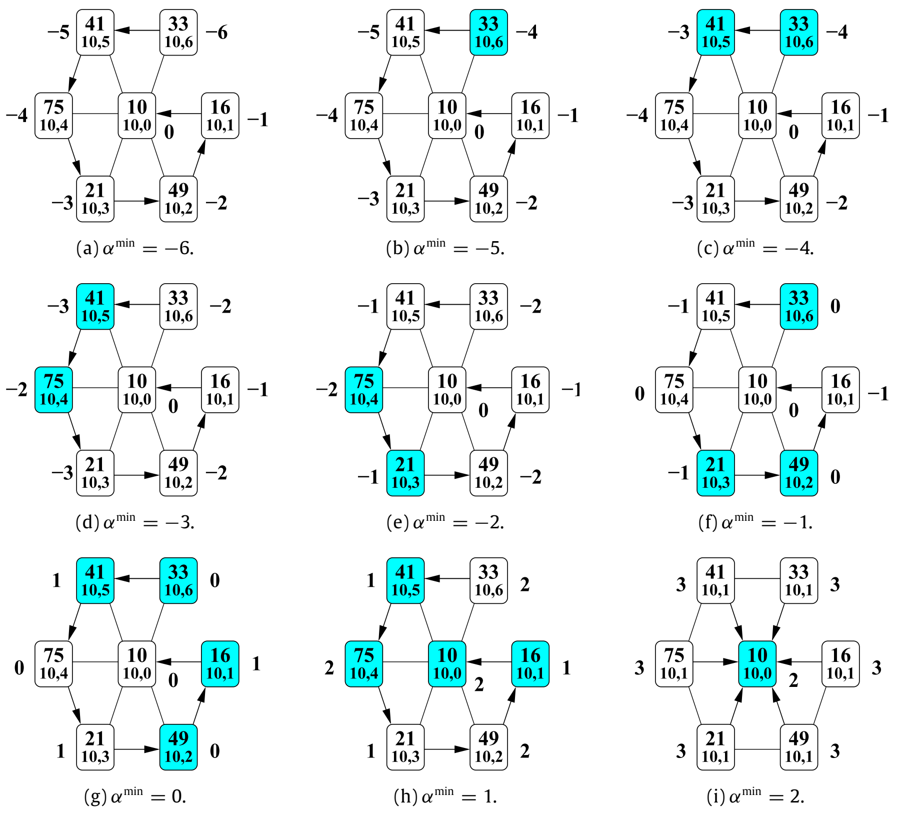


Fig. 6.4. If Benchmark BM4 holds, the value of $\alpha^{\min}$ increases every round until Benchmark BM5 holds. In the worst case, Benchmark BM5 is reached in $\Theta(n)$ rounds.

6.1.7. Benchmark BM7

Assume Benchmark BM6 holds. The energy of $\mathbb{L}$ cannot exceed $2n$, and cannot increase, because $\mathbb{L}$ is *color frozen*, and thus cannot execute a color action. The *color* waves will continue to be generated at the leaves and move toward $\mathbb{L}$, but they cannot pass each other. After $O(diam)$ rounds, the network will be *color locked*, and no further color actions can be enabled. Since $P.done = Done(P)$ for all P, Action D5 cannot execute, either. Thus, no process can execute any action, and silence has been achieved.

6.2. The unfair daemon

Even if all benchmarks are achieved in finitely many rounds, there is no guarantee that the computation will ever end. If a process P is enabled, the unfair daemon is never required to select P, unless P is the only enabled process. If there is always some other enabled process, then P could be enabled forever, and the last round would continue forever.

This cannot happen for SSLE. Given an initial configuration γ_0, there are only finitely many configurations that are reachable from γ_0. Any infinite computation must repeat a configuration, and thus contains a cycle, *i.e.*, a computation that begins and ends at the same configuration.

Suppose $Z :: \gamma_0 \mapsto \cdots \mapsto \gamma_m = \gamma_0$ is a cycle, for $m > 0$. We show that Z cannot contain any action, and hence cannot exist.

Let $\mathcal{K}$ be the set of all processes that execute a structure action during Z. We make a choice of $P \in \mathcal{K}$ and γ_i which minimizes $P.key$, *i.e.*, we pick $P \in \mathcal{K}$ and $0 < i \le m$ such that the value of $P.key$ at γ_i is less than or equal to the value of $Q.key$ at γ_j for any $Q \in \mathcal{K}$ and any $0 < j \le m$. We also insist that P executes a structure action at the step $\gamma_{i-1} \mapsto \gamma_i$. By minimality, $P.key$ decreases during the step, which means that the structure action must be $Attach(Q)$ for some Q. If $Q \in \mathcal{K}$, that contradicts the minimality of P, while if $Q \notin \mathcal{K}$, $P.key$ will be unable to increase its value during the cycle, contradiction. We conclude that $\mathcal{K} = \emptyset$, and thus Z contains no structure action.

Let $\mathcal{F}$ be the set of all processes that execute a color action during Z. We can also show that $\mathcal{F} = \emptyset$. We will only sketch the proof here. During Z, components cannot change since no process executes a structure action. Each $P \in \mathcal{F}$ must execute at least one of each of the two color actions during Z, and thus $P.parent \in \mathcal{F}$ and $True\_Chldrn(P) \subseteq \mathcal{F}$. It follows that $\mathcal{F}$ is the exact union of components. None of these can be a false tree, since a false root cannot execute a color action. Pick one of those components, $\mathcal{C}$. If $Recruits(P) \neq \emptyset$ for some $P \in \mathcal{C}$, then P can never execute Action D3, contradiction. If $Recruits(P) = \emptyset$

for all $P \in \mathcal{C}$, then eventually $P.done$ will be true for all $P \in \mathcal{C}$, and the root of $\mathcal{C}$ will be color frozen, contradiction. Thus, $\mathcal{F} = \emptyset$.

The only remaining possible action during Z is D5. Let $\mathcal{D}$ be the set of all processes that Action D5 during Z. Each $P \in \mathcal{D}$ must change $P.done$ at least twice during the cycle, which means that $Done(P)$ must be sometimes true and sometimes false. Pick $P \in \mathcal{D}$ such that $P.level \geq Q.level$ for all $Q \in \mathcal{D}$. Then, the value of $Done(P)$ cannot change during the cycle, contradiction.

We conclude that no cyclic computation can exist, and thus there is no infinite computation of SSLE. When the computation ends, SSLE is silent.

7. Formal proofs

7.1. Proofs of benchmarks

7.1.1. Benchmark BM1

Lemma 7.1. *Benchmark BM1 is closed, and holds within one round of initialization.*

Proof. BM1 is closed since there is no action of a process P that can cause $P.key$ to exceed $Self\_Key(P)$.

If $P.key > Self\_Key(P)$, then P will be enabled to execute Action D2. That action cannot become neutralized by any action of any other process, and hence P will reset (*i.e.*, execute Action D2) within one round. □

7.1.2. Benchmark BM2

Recall that a process P is *subfinal* if $P.key < Final\_Key(P)$. Thus, Benchmark BM2 simply states that no process is subfinal.

Lemma 7.2. *Benchmark BM2 is closed.*

Proof. The only way that a process which is not subfinal can become subfinal is to attach to a process which is already subfinal. Thus, once BM2 holds, it must continue to hold. □

We now give a formal recursive definition of *energy*, which agrees with the informal definition given in Section 6.1.2[1] and prove several lemmas.

We say that P is a *leaf* if $True\_Chldrn(P) = \emptyset$. Recursively define $E(P)$, the *energy* of any process P, and E_I for any ID I, as follows.

- For processes P and Q, let $\Delta(P,Q) = \begin{cases} 1 & \text{if } P.color \neq Q.color \\ 2 & \text{if } P.color = Q.color. \end{cases}$
- $E(P) = \begin{cases} 1 & \text{if } P.color = 1 \text{ and } P \text{ is a leaf} \\ 2 & \text{if } P.color = 2 \text{ and } P \text{ is a leaf} \\ \max\{E(Q) + \Delta(P,Q) : Q \in True\_Chldrn(P)\} & \text{otherwise.} \end{cases}$
- For any I of ID type (not necessarily the ID of any process in the network) let
  - $E_I = \max\{E(P) : P.leader = I\}$,
  - $N_I = |\{E(P) : P.leader = I\}|$, the number of processes whose *leader* variable is I.

By the definition of energy, and since there are n processes altogether, we have:

Remark 7.3. Suppose Benchmark BM1 holds.
(a) For any process P, $E(P) \geq 1$, and if $P.color = 2$, then $E(P) \geq 2$.
(b) For any I, $E_I \leq 2N_I \leq 2n$.
(c) For any $I < \mathbb{L}.id$, $E_I \leq 2n - 2$.

"Prime" notation. In Lemma 7.4 below, and in many subsequent lemmas in this section, we consider a specific step $\gamma \mapsto \gamma'$, and we will use "prime" notation to indicate the values of variables and functions at γ', while notation without the "prime" will refer to the values at γ. For example, the values of $P.color$ at γ and at γ', will be written as $P.color$ and $P.color'$, respectively.

Lemma 7.4. *Suppose Benchmark BM1 holds. Let $\gamma \mapsto \gamma'$ be one step, and suppose that P does not execute Action D2 during that step. Then*

(a) $E'(P) \leq E(P) + 1$,
(b) *if P does not execute a color action during the step, then $E'(P) \leq E(P)$.*

Proof. We prove both statements simultaneously by backwards induction on $P.level'$.

[1] The definition of energy has been changed from the conference paper.

Case I: P is a leaf at γ'. If P executes Action D4 during the step, then $E'(P) = 2 \le E(P) + 1$, by Remark 7.3. Otherwise, either $P.color' = 1$ or $P.color' = P.color$, and thus $E'(P) \le E(P)$.

Case II: P is not a leaf at γ'. Then P cannot have executed Action D1 during the step. Pick $Q \in True\_Chldrn'(P)$ such that $E'(P) = \Delta'(P, Q) + E'(Q)$. Since Q is a true child at γ', it cannot have executed Action D2 during the step. We break this case into four subcases:

Case II(a): Q executes Action D1 during the step, attaching to P. By the guards of Action D1 and the color actions, $P.color = P.color' = 2$. By the definition of energy, $E'(Q) = 1$ and thus $E'(P) = 2 \le E(P)$.

Case II(b): Q executes a color action during the step. By the guards of the color actions, P cannot have executed a color action during the step. $\Delta(P, Q) = 2$, $\Delta'(P, Q) = 1$, and by the inductive hypothesis, $E'(Q) \le E(Q) + 1$. Thus $E'(P) = E'(Q) + 1 \le E(Q) + 2 \le E(P)$.

Case II(c): P executes a color action during the step. By the guards of the color actions, Q cannot have executed a color action during the step. $\Delta(P, Q) = 1$, $\Delta'(P, Q) = 2$, and by the inductive hypothesis, $E'(Q) \le E(Q)$. Thus $E'(P) = E'(Q) + 2 \le E(Q) + 2 \le E(P) + 1$.

Case II(d): Neither P nor Q executes a color action during the step, and Q does not execute Action D1 during the step. Then $\Delta(P, Q) = \Delta'(P, Q)$, and by the inductive hypothesis, $E'(Q) \le E(Q)$. Thus $E'(P) = E'(Q) + \Delta'(P, Q) \le E(Q) + \Delta(P, Q) \le E(P)$. □

Lemma 7.5. *Suppose Benchmark BM1 holds. If $I < \mathbb{L}.id$, then E_I cannot increase.*

Proof. Consider one step, $\gamma \mapsto \gamma'$, and suppose that $E'_I > E_I$. If $E_I = 0$, then $N_I = 0$, *i.e.*, there are no processes whose *leader* is I. Since there is also no process whose ID is I, there is no way that there can ever again be a process whose *leader* is I; thus $E'_I = 0$, contradiction.

Thus, $E'_I \ge 2$. Pick a process P such that $P.leader' = I$ and $E'(P) = E'_I$. If P executes Action D2 during the step, then P is not subfinal at γ', contradiction. If P executes Action D1 during the step, then $E'(P) = 1$, contradiction.

In all other cases, $P.leader = I$. If P does not execute a color action, then, by Lemma 7.4(b), $E'_I = E'(P) \le E(P) \le E_I$, contradiction. If P executes a color action, then P must be a true child at γ, since P cannot be a true root and false roots cannot execute color actions. Let $Q = P.parent$; then $Q.leader = I$. By the definition of energy and by Lemma 7.4(a), $E'_I = E'(P) \le E(P) + 1$ and $E_I \ge E(Q) \ge E(P) + 1$, contradiction. □

We need a rather non-intuitive technical lemma.

Lemma 7.6. *Suppose Benchmark BM1 holds. Let $\gamma \mapsto \gamma'$ be a step. Suppose $P.leader' = I < \mathbb{L}.id$, and either $P.leader \ne I$ or $E(P) < E_I$. Then $E'(P) < E_I$.*

Proof. P cannot execute Action D2 during the step, since then $P.leader' \ge \mathbb{L}.id$. If P executes Action D1 during the step, attaching to some process Q, then $Q.leader = I$ and $Q.color = 2$. Thus, $E'(P) = 1 < 2 \le E(Q) \le E_I$.

If P executes a color action during the step, then $P.leader = I$ and $P.parent.leader = I$. Thus $E_I \ge E(P.parent) \ge E(P) + 2 \ge E'(P) + 1$, where the last inequality is from Lemma 7.4(a).

If P does not execute any structure or color action during the step, then $P.leader = I$, and by Lemma 7.4(b). $E'(P) \le E(P) < E_I$. □

Lemma 7.7. *Suppose Benchmark BM1 holds. If $I < \mathbb{L}.id$ and $E_I > 0$, then E_I decreases within one round.*

Proof. Suppose that $\gamma_0 \mapsto \cdots \mapsto \gamma_m$ is a round $E_I > 0$ at γ_0, and E_I does not decrease during that round. By Lemma 7.5, E_I must be constant throughout that round. We use the notation that a subscript i on a quantity indicates the value of that quantity at γ_i.

Pick P such that $P.leader_m = I$ and $E_m(P) = E_I$. Applying Lemma 7.6 m times in succession, we have that $P.leader_i = I$ and $E_i(P) = E(I)$ for $i = m - 1, \ldots, 0$. Thus, P does not execute any structure action at any step in the round.

P must be a false root for all γ_i, since if P were a true child of a process Q, then $Q.leader_i = I$ and $E_i(Q) > E_I$. Thus, P is enabled to execute a structure action at every configuration in the round, which contradicts the definition of round. □

Lemma 7.8. *If Benchmark BM1 holds, Benchmark BM2 will hold within $2n - 2$ rounds.*

Proof. For any $I < \mathbb{L}.id$, E_I cannot exceed $2n - 2$, and by Lemma 7.7, $E_I = 0$ within $2n - 2$ rounds. Thus, there are no remaining subfinal processes. □

7.1.3. Benchmark BM3

Recall that a process is *pseudo-final* if $P.key = Final\_Key(P)$, and $P \notin Final$.

Lemma 7.9. *Benchmark BM3 is closed.*

Proof. Given that Benchmark BM2 holds, The only way that a process that is not pseudo-final can become pseudo-final is to attach to another process that is already pseudo-final. Thus, once Benchmark BM3 holds, it will hold forever. □

$$\text{Define } \Pi = \begin{cases} \infty & \text{if Benchmark BM3 holds} \\ \min\{P.level : P \text{ is pseudo-final}\} & \text{otherwise.} \end{cases}$$

Lemma 7.10. *Suppose Benchmark BM3 holds. If $\Pi < \infty$, then Π increases during the next round.*

Proof. If P is not pseudo-final and $||P, \mathbb{L}|| \le \Pi$, then P cannot become pseudo-final, because it could only do so by attaching to a pseudo-final process that is closer to $\mathbb{L}$. On the other hand, if P is pseudo-final and $||P, \mathbb{L}|| = \Pi$, then P is a false root, and is thus enabled to execute Action D1 or D2, and will do so within one round. Thus, by the end of that round, Π must increase. □

Lemma 7.11. *If Benchmark BM2 holds, then Benchmark BM3 holds within diam rounds.*

Proof. Initially, $\Pi \ge 1$. By Lemma 7.10, after *diam* rounds, $\Pi \ge diam + 1$. Since the maximum finite value of Π is *diam*, there can then be no pseudo-final processes. □

7.1.4. Evolution of $Component\_Tree(\mathbb{L})$ and Benchmark BM4

In this section, we discuss the evolution of the final BFS tree itself. As soon as Benchmark BM2 holds, $\mathbb{L}$ is the root of a component which eventually grows into the final BFS tree.

Lemma 7.12. *If Benchmark BM3 holds and $P \in Final$, then P will never again execute a structure action.*

Proof. If $Q \in \mathcal{N}_P$ and $Q.key = Best\_Nbr\_Key(P)$ then Q is subfinal, contradiction; thus P cannot execute D1. Since P is not a false root, it cannot execute D2. □

Corollary 7.13. *If Benchmark BM3 holds, and if a process is final, it will remain final.*

Lemma 7.14. *If Benchmark BM3 holds, $P \in Final$, and $P \notin Active$, then P will never again become active.*

Proof. Since Benchmark BM3 holds, if Q is final, then $Recruits(Q)$ consists exactly of the non-final neighbors of Q. Thus, since P is not active, every neighbor of every descendant of P is final. By Corollary 7.13, those neighbors remain final, and the descendants of P can acquire no new children. □

We prove Benchmark BM4 by backwards induction on the distance of a process to $\mathbb{L}$.

Lemma 7.15. *If at least t rounds have elapsed since Benchmark BM3 first held, and if P is active and $P.level + t > diam$, then $P.done$ is false.*

Proof. By induction on t. If $t = 0$, we are done trivially, since $P.level = ||P, \mathbb{L}|| \le diam$.

We prove the inductive step by contradiction. Suppose $t > 0$, and $P.done$ holds. Let γ_p be the first configuration at which both P is final and $(t - 1)$ rounds have elapsed since Benchmark BM3 first held. and let γ_q be the current configuration. By Corollary 7.13 and Lemma 7.14, P is active at every configuration between γ_p and γ_q, inclusive. Thus, by the recursive definition of *Active*, for each $p \le i \le q$, there is some $Q \in \mathcal{N}_P$ such that either Q is not final at γ_i, or $||Q, \mathbb{L}|| = ||P, \mathbb{L}|| + 1$ and Q is active at γ_i. In the latter case, by the inductive hypothesis, $Q.done$ is false at γ_i.

Thus, at every configuration in the interval between γ_p and γ_q, $Done(P)$ is false. It follows that $P.done$ cannot become false. If that interval includes a complete round, then, since P is enabled to execute either a color action or Action D5 during that entire round, it must execute, changing $P.done$ to false, contradiction. Otherwise, γ_p is the first configuration at which P is final, which implies that P executes a structure action at that step, changing $P.done$ to false. In either case, $P.done$ must be false at γ_q. □

Corollary 7.16. *If Benchmark BM3 holds, then Benchmark BM4 holds within $diam + 1$ rounds.*

7.1.5. Benchmark BM5

We remind the reader that, if Benchmark BM2 holds, then by Corollary 7.13, once $P \in$ *Final*, it can never leave *Final*. We will use this fact repeatedly in subsequent lemmas without specifically referring to that corollary. Let T be the number of steps of the computation until Benchmark BM5 holds; if BM5 never holds,[2] we let $T = \infty$. For any integer $0 \le t \le T$, let γ_t be the configuration t steps after γ_0. We use a subscript of t to denote the value of any quantity at configuration γ_t.

Recall the function α on $Component\_Tree(\mathbb{L})$, which we defined in Section 6.1.5.

Remark 7.17. If $P \in Component\_Tree(\mathbb{L})$, then $(\alpha(P) + P.level) \bmod 4 = \begin{cases} 0 & \text{if } P.color = 1 \\ 2 & \text{if } P.color = 2. \end{cases}$

Note that $\alpha(P)$ could be positive, negative or zero. In fact:

Remark 7.18. $-n + 1 \le \alpha(P) \le n + 1$.

Lemma 7.19. *Let $0 < t \le T$. Suppose $P \in Component\_Tree_t(\mathbb{L})$ and P does not execute a structure action during the step $\gamma_{t-1} \mapsto \gamma_t$. Then*

(a) $P \in Component\_Tree_{t-1}(\mathbb{L})$.

(b) $\alpha_t(P) = \begin{cases} \alpha_{t-1}(P) + 2 & \text{if } P \text{ executes a color action during the step} \\ \alpha_{t-1}(P) & \text{otherwise.} \end{cases}$

Proof. We prove (a) by induction on $P.level$. If $P.level_t = 0$, then $P = \mathbb{L}$, and we are done.

Suppose $P.level_t > 0$. Let $Q = P.parent_t$. Since P did not execute a structure action during the step, $P.parent_{t-1} = Q$. If $Q = \mathbb{L}$, we are done. Assume $Q \ne \mathbb{L}$, Q could not have executed Action D2 during the step, since it would then not be in $Component\_Tree_t(\mathbb{L})$. Q could also not have executed Action D1 during the step, since that action would change $Q.key$, which would imply that $P \in False\_Chldrn_{t-1}(Q)$, thus disabling Action D1.[3] By the inductive hypothesis, $Q \in Component\_Tree_{t-1}(\mathbb{L})$, and thus $P \in Component\_Tree_{t-1}(\mathbb{L})$.

We now prove (b), also by induction on $P.level$. If $P.level_t = 0$, then $P = \mathbb{L}$, and we are done by the definition of $\alpha(\mathbb{L})$. Otherwise, let $Q = P.parent_t = P.parent_{t-1}$. By (a), P, Q are both in $Component_{t-1}$. We break the proof into three cases and use the inductive hypothesis and the definition of $\alpha(P)$ in terms of $\alpha(Q)$ in each case.

Case I: Neither P nor Q executes a color action during the step $\gamma_{t-1} \mapsto \gamma_t$. Then $\alpha_t(Q) = \alpha_{t-1}(Q)$ by the inductive hypothesis, and $\alpha_t(P) - \alpha_t(Q) = \alpha_{t-1}(P) - \alpha_{t-1}(Q)$ by the definition of $\alpha(P)$.

Case II: P executes a color action during the step, and Q does not. By the inductive hypothesis, $\alpha_t(Q) = \alpha_{t-1}(Q)$, while $\alpha_t(P) = \alpha_t(Q) + 1$ and $\alpha_{t-1}(P) = \alpha_{t-1}(Q) - 1$ by the definition of $\alpha(P)$ and the second clause of the guard of the color action.

Case III: Q executes a color action during the step, and P does not. By the inductive hypothesis, $\alpha_t(Q) = \alpha_{t-1}(Q)$, while $\alpha_t(P) = \alpha_t(Q) - 1$ and $\alpha_{t-1}(P) = \alpha_{t-1}(Q) + 1$ by the definition of $\alpha(P)$ and the third clause of the guard of the color action. $\square$

Lemma 7.20. *$\alpha^{\min}$ does not decrease.*

Proof. By Lemma 7.19, the only way that $\alpha(P)$ can decrease is for P to execute Action D1. Suppose that $\alpha^{\min} = \alpha(P)$, and that $\alpha^{\min}$ decreased during the last round. If P just executed D1 , attaching to a process $Q \in Component\_Tree(\mathbb{L})$, then $\alpha(Q) = \alpha(P) - 1$, hence $\alpha(P) > \alpha^{\min}$. Otherwise, by Lemma 7.19, $\alpha(P)$ did not decrease during the round, and therefore $\alpha^{\min}$ did not decrease. Either way, we have a contradiction. $\square$

Lemma 7.21. *Starting with any configuration γ_p for $0 \le p < T$, either $\alpha^{\min}$ increases, or γ_T is reached, within two rounds.*

Proof. Suppose that, within two rounds, $\alpha^{\min}$ does not increase and γ_T is not reached. Let $\mathcal{M} = \{P \in Component\_Tree(\mathbb{L}) : \alpha(P) = \alpha^{\min}\}$.

We claim that, during the two rounds, no process may join $\mathcal{M}$. Suppose $P \in \mathcal{M}_t$ for some $t > p$. P cannot have executed D2 during the step $\gamma_{t-1} \mapsto \gamma_t$, since it would then not be in $Component\_Tree_t$. If P executed D1 during the step, attaching to some process Q, then $\alpha_t(P) = \alpha_{t-1}(Q) + 1 \ge \alpha^{\min} + 1$, contradiction. Otherwise, by Lemma 7.19, $P \in \mathcal{M}_{t-1}$.

If $P \in \mathcal{M}_p$ and $P.color_p = 1$, then P is enabled to execute D1 or D4. During the first round, P will execute one of these actions, leaving $\mathcal{M}$.

If $P \in \mathcal{M}_p$ and $p.color_p = 2$, then every member of $Recruits(P)$ will execute D1 during the first round. Thus, if P is still in $\mathcal{M}$, it will be enabled to execute either D1 or D3 during the next round, and will therefore leave $\mathcal{M}$ within two rounds of γ_p.

Since $\mathcal{M} = \emptyset$ after two rounds have been completed, $\alpha^{\min}$ must have increased, contradiction. $\square$

For any process P, define the set of *predecessors* of P to be $Preds(P) = \{Q \in \mathcal{N}_P : ||Q, \mathbb{L}|| + 1 = ||P, \mathbb{L}||\}$. For any $P \ne \mathbb{L}$, $Preds(P)$ is the set of all processes which might be chosen to be $P.parent$ in the BFS tree rooted at $\mathbb{L}$.

2 As we shall see, that is impossible.

3 This is the only place were we need the third clause in the guard of D1.

Lemma 7.22. *Let P be a process, and let $h = ||P, \mathbb{L}||$.*
(a) *If $Q \in Preds(P)$, $Q \in Final_i$, and $P \notin Final_i$ for some $0 \le i < T$, then $\alpha_i(Q) \le 3h - 1$.*
(b) *For some $0 \le j \le T$, $P \in Final_j$ and $\alpha_j(P) \le 3h + 2$.*

Proof. By induction on h. If $h = 0$, then $P = \mathbb{L}$, and (a) is vacuously true. For (b), pick $j = 0$; we are done since $\alpha_0(\mathbb{L}) \in \{0, 2\}$.

Suppose $h > 0$. We first prove (a). By the inductive hypothesis, part (b), there exists j such that $Q \in Final_j$ and $\alpha_j(Q) \le 3h - 1$. If $\alpha_i > 3h - 1$, then, by Lemma 7.19 and Remark 7.17, there exists k, where $j \le k < i$, $\alpha_k = 3h - 1$, and $Q.color_k = 2$. By picking the largest such k, we can insist that $\alpha_{k+1}(Q) = 3h + 1$, which implies that Q executes Action D3 during the step $\gamma_k \mapsto \gamma_{k+1}$. However, the guard of D3 implies that $P.key_k \le Succ\_Key(Q) = (\mathbb{L}.id, h)$. Since Benchmark BM2 holds, $P.key_k = (\mathbb{L}.id, h)$, *i.e.,* $P \in Final_k \subseteq Final_i$, contradiction.

We now prove (b). If $P \in Final_0$, then let $j = 0$. The chain of processes, following the parent pointers from P to $\mathbb{L}$, has length h, and all processes in that chain are in $Final_0$. Thus $\alpha_0(P) \le h + 2$ by the definition of α, and we are done.

Consider now the case $P \notin Final_0$. By the inductive hypothesis, $Preds(P) \subseteq Final$ eventually. Pick the smallest t such that $Preds(P) \cap Final_t$. Also by the inductive hypothesis, $\alpha_t(Q) \le 3h - 1$ for all $Q \in Preds(P) \cap Final_t$. By Lemma 7.19, $\alpha(Q)$ cannot decrease if $Q \in Final$. Pick the smallest j such that $\alpha_j(Q) > 3h - 1$ for some $Q \in Preds(P) \cap Final_j$. By Lemma 7.19, $\alpha_j(Q) \le 3h + 1$ for all $Q \in Preds(P) \cap Final_j$, and is equal to $3h + 1$ for at least one such Q. By (a), $P \in Final_j$. Then $\alpha_j(P) \le 3h + 2$ by the definition of α, since $P.parent_j \in Preds(P) \cap Final_j$; and we are done. □

Lemma 7.23. *If Benchmark BM4 holds, then Benchmark BM5 holds within $2n + 6\,diam + 4$ rounds.*

Proof. Let P be any process, and let $h = ||P, \mathbb{L}|| \le diam$. By Lemma 7.22, there is some $j \le T$ such that $P \in Final_j$ and $\alpha_j(P) \le 3\,diam + 2$. By Remark 7.18, $\alpha_0^{\min} \ge -n + 1$. By Lemma 7.21, if there were $2n + 6\,diam + 4$ complete rounds between γ_0 and γ_j, $\alpha_j^{\min}$ would exceed $3\,diam + 2$, contradiction. Thus, $P \in Final$ within $2n + 6\,diam + 4$ rounds. □

7.1.6. Benchmarks BM6 and BM7

Lemma 7.24. *Benchmark BM6 is closed, and if Benchmark BM5 holds, Benchmark BM6 will hold within $diam + 1$ rounds.*

Proof. If Benchmark BM6 holds, then $Done(P)$ for all P, so $P.done$ must remain true for every P. Thus, BM6 is closed.

Suppose Benchmark BM5 holds. Let $\ell = \max\{P.level : \neg P.done\}$. If $P.level = \ell$ and $\neg P.done$, then $Done(P)$, and P is thus enabled to execute either a color action or Action D5, and will do so in the next round, changing $P.done$ to true. Thus, within $diam + 1$ rounds, $P.done$ for all P. □

Lemma 7.25. *Benchmark BM7 is closed, and if Benchmark BM6 holds, Benchmark BM7 will hold within $2\,diam - 1$ rounds.*

Proof. Define an integral function η on processes as follows:

- $\eta(\mathbb{L}) = 0$
- If $P \ne \mathbb{L}$ then $\eta(P) = \begin{cases} \eta(P.parent) - 1 & \text{if } P.color = P.parent.color \\ \eta(P.parent) + 1 & \text{if } P.color \ne P.parent.color. \end{cases}$

Let $\mathcal{H} = \{P \ne \mathbb{L} : (P.parent.color = P.color) \wedge (\forall Q \in Chldrn(P) : Q.color \ne P.color)\}$, the set of local minima of η other than $\mathbb{L}$. Then $\mathcal{H} = \emptyset$ if and only if Benchmark BM7 holds.

Since Benchmark BM6 holds, a process P is enabled to execute a color action if and only if $P \in \mathcal{H}$. Thus, if $P \in \mathcal{H}$, P is enabled to execute a color action, and will do so within one round.

Let $h = \min\{\eta(P) : P \in \mathcal{H}\}$. By the definition of η, either $\mathcal{H} = \emptyset$ or $-diam \le h \le diam - 2$. The value of h will increase every round until $\mathcal{H}$ is empty, and thus $\mathcal{H} = \emptyset$ within $2\,diam - 1$ rounds. □

7.2. The unfair daemon

In this section, we show that SSLE works under the unfair daemon, meaning that every round must eventually end.

Our method is by contradiction. If SSLE does not work under the unfair daemon, there must exist an execution of SSLE of infinite length. Since there are only finitely many configurations reachable from any given start configuration, that implies that there must exist a cyclic execution, meaning, an execution of positive length that begins and ends at the same configuration. We will then prove that a cyclic execution cannot contain any action, and therefore cannot exist.

We first need some new notation.

- For any process P, define $E\_plus\_L(P) = E(P) + P.level$, the *energy plus level* of P.
- $E\_plus\_L_{upp\_bnd} = \max\{2n, \max_P\{E\_plus\_L(P)\}\}$, an upper bound on $E\_plus\_L(P)$ for all P.

Lemma 7.26. *The value of $E\_plus\_L_{upp\_bnd}$ cannot increase.*

Proof. Let $\gamma \mapsto \gamma'$ be any step of SSLE. We will prove that $E\_plus\_L'_{upp\_bnd} \le E\_plus\_L'_{upp\_bnd}$. Suppose that $E\_plus\_L_{upp\_bnd} > E\_plus\_L_{upp\_bnd}$. Then, there is some process P such that $E\_plus\_L'(P) > E\_plus\_L_{upp\_bnd}$.

If P executed action D1 during the step, attaching to a process Q, then $E\_plus\_L'(P) < E\_plus\_L(Q) \le E\_plus\_L_{upp\_bnd}$, contradiction. If P executed action D2 during the step, then $E\_plus\_L'(P) = 2 < E\_plus\_L_{upp\_bnd}$, contradiction. If P executed a color action during the step, and P was a true root at γ, then $E\_plus\_L'(P) \le 2n \le E\_plus\_L_{upp\_bnd}$, contradiction. If P executed a color action during the step, and P was a true child at γ, let $Q = P.parent$. Then $E\_plus\_L'(P) < E\_plus\_L(Q) \le E\_plus\_L_{upp\_bnd}$, contradiction. □

Reachable configurations. We say that a configuration γ' is *reachable from* a given configuration γ if there is some execution of SSLE that begins with γ and contains γ'.

Lemma 7.27. *Given a configuration γ of a network, there are at most finitely many configurations of the network that are reachable from γ.*

Proof. We need to show that each of the variables of every process has a finite range.

1. For any reachable configuration, the value of $P.parent$ must equal either the initial value of $P.parent$ or $Q.id$ for some process Q, and the value of $P.leader$ must equal either $Q.id$, or the initial value of $Q.leader$ for some process Q. Since there are only n processes, there at most $n + 1$ possible values for $P.parent$ and at most $2n$ possible values for $P.leader$.
2. By Lemma 7.26, no value of $P.level$ in any configuration reachable from γ can exceed the value of $E\_plus\_L_{upp\_bnd}$ at γ; hence there are only finitely many values of $P.level$.
3. The number of possible values of $P.color$ is two.
4. The number of possible values of $P.done$ is two.

Thus, the number of possible states of P, over all configurations reachable from γ is finite. Since the network is finite, the number of configurations reachable from γ is finite. □

We define a *cyclic computation* (or simply *cycle*) to be an computation of positive length which starts and ends at the same configuration.

Lemma 7.28. *Any infinite computation of* SSLE *contains a cycle.*

Proof. By Lemma 7.27 and the pigeonhole principle. □

Henceforth in this section, let $Z : \gamma_0 \mapsto \gamma_1 \mapsto \cdots \mapsto \gamma_m = \gamma_0$, for $m > 0$, be a cycle.

Lemma 7.29. *Z contains no action which changes $P.key$ for any process P.*

Proof. Our proof is by contradiction; suppose the statement of the lemma is false. Let $\mathcal{K}$ be the set of all processes P such that $P.key$ changes during the cycle Z.

For $P \in \mathcal{K}$, define $Min\_Key(P)$ to be the minimum value of $P.key$ over all configurations in the cycle Z. Since $P.key$ is not constant, $P.key > Min\_Key(P)$ at at least one configuration in Z.

Let $k = \min \{Min\_Key(P) : P \in \mathcal{K}\}$. We first note that $P.key \le Self\_Key(P)$ for all $P \in \mathcal{K}$ at all configurations in the cycle, since there is no action which can set $P.key > Self\_Key(P)$.

Case I: $k = Self\_Key(P)$ for some $P \in \mathcal{K}$. Then $P.key > Self\_Key(P)$ at some configuration in the Z, contradiction.

Case II: $k < Self\_Key(P)$ for all $P \in \mathcal{K}$. Pick $P \in \mathcal{K}$ such that $Min\_Key(P) = k$. By hypothesis, Z must contain a step at which $P.key$ is set to k. At that step, P executes Action D1 (Q), selecting some process Q to be its parent. Since $Q.key < k$, $Q \notin \mathcal{K}$, and $Q.key$ cannot change during the cycle. P is then never enabled to increase its key, since $P.parent = Q$ forever, contradiction. □

Lemma 7.30. *During Z, P.done cannot change for any process P.*

Proof. Our proof is by contradiction. Let P be the process of maximum key such that $P.done$ changes during the cycle. No process executes a structure action during the cycle, since otherwise its key would change, contradicting Lemma 7.29. Thus, the sets $Recruits(P)$ and $Chldrn(P)$ do not change. Since $Q.done$ does not change for any $Q \in Chldrn(P)$, the value of $Done(P)$ does not change. Thus, $P.done$ can change at most once during the cycle, contradiction. □

Lemma 7.31. *During Z, P.color cannot change for any process P.*

Proof. Our proof is by contradiction. Let $\mathcal{F}$ be the set of all processes P for which $P.color$ changes during Z.

If $P \in \mathcal{F}$, then P must change color at least twice during the cycle. Thus, $P.parent$ must change color, and all $Q \in Chldrn(P)$ must also change color, during the cycle. It follows that $Component\_Tree(P) \subseteq \mathcal{F}$.

We claim that $P.done$ holds for all $P \in \mathcal{F}$. Suppose not. Let P be the member of $\mathcal{F}$ of maximum key such that $\neg P.done$. $Recruits(P) = \emptyset$, since otherwise P would not be enabled to execute Action D3. $Q.done$ for all $Q \in Chldrn(P)$, by the maximality of $P.key$. Thus, $Done(P)$ holds, which will cause $P.done$ to be set to true when P executes a color action, contradiction.

Since $\mathcal{F}$ contains an entire component, it must contain a process R which is the root of that component. If R is not a true root, then R is not enabled to execute a color action. If R is a true root, then $R.done$ holds, and thus R is color frozen, implying that R cannot execute a color action. In either case, $R \notin \mathcal{F}$, contradiction. □

Lemma 7.32. *Every computation of* SSLE *is finite.*

Proof. Suppose there is an infinite computation of SSLE. By Lemma 7.28, every infinite computation of SSLE contains a cycle, Z. Z must contain at least one action. But, by Lemmas 7.30 and 7.31, and 7.29, Z can contain no action, contradiction. □

7.3. Main result

Theorem 7.33. SSLE *converges within $O(n)$ rounds under the unfair daemon, and is silent with $O(diam)$ additional rounds.*

Proof. By Lemma 7.1, Benchmark BM1 holds within one round of an arbitrary initial configuration.

By Lemma 7.8, Benchmark BM2 holds within $2n - 2$ additional rounds.

By Lemma 7.11, Benchmark BM3 holds within *diam* additional rounds.

By Corollary 7.16, Benchmark BM4 holds within $diam + 1$ additional rounds.

By Lemma 7.23, Benchmark BM5 holds within $2n + 6\,diam + 4$ additional rounds. Thus, starting from an arbitrary configuration, the network is in a legitimate configuration in at most $4n + 8\,diam + 4$ rounds.

By Lemma 7.24, Benchmark BM6 will hold within $diam + 1$ additional rounds.

By Lemma 7.25, Benchmark BM7 will hold within $2\,diam - 1$ additional rounds.

When Benchmark BM7 holds, no process is enabled, and thus, starting from an arbitrary configuration, SSLE converges and is silent within $4n + 11\,diam + 4$ rounds.

We now consider the problem of the unfair daemon. Even an algorithm which takes finitely many rounds could execute forever if the unfair daemon never selects an enabled process, causing a round to never end. But, by Lemma 7.32, no execution of SSLE can last forever. Thus, SSLE converges and is silent under the unfair daemon. □

8. Conclusion

We present a uniform silent self-stabilizing asynchronous distributed algorithm, SSLE, for election of a leader of a network, in which processes have unique IDs. The algorithm stabilizes in $O(n)$ rounds, using $O(\log n)$ space per process, and becomes silent after an additional $O(diam)$ rounds, under the unfair daemon.

References

[1] Y. Afek, A. Bremler, Self-stabilizing unidirectional network algorithms by power-supply, in: 8th Annual ACM Symposium on Discrete Algorithms, 1997, pp. 111–120.

[2] A. Arora, M.G. Gouda, Distributed reset, IEEE Transactions on Computers 43 (1994) 1026–1038.

[3] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, Time optimal self-stabilizing synchronization, in: 25th Annual ACM Symposium on Theory of Computing, 1993, pp. 652–661.

[4] A.K. Datta, L.L. Larmore, P. Vemula, Self-stabilizing leader election in optimal space, in: 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2008, Detroit, Michigan, November 21–23, in: Lecture Notes in Computer Science, vol. 5340, Springer-Verlag, 2008, pp. 109–123.

[5] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Communications of the Association for Computing Machinery 17 (1974) 643–644.

[6] S. Dolev, Self-Stabilization, The MIT Press, Cambridge, 2000.

[7] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, Chicago Journal of Theoretical Computer Science 1997-4 (1997) 1–40.

[8] M.G. Gouda, N. Multari, Stabilizing communication protocols, IEEE Transactions on Computing 40 (1991) 448–458.