Self-Stabilizing Silent Disjunction in an Anonymous Network *

Ajoy K. Datta Stéphane Devismes Lawrence L. Larmore

Abstract

In this paper, we give a distributed silent self-stabilizing algorithm, DISJ, for the *disjunction problem* in a connected network. In this problem, each process x has an input bit $x.in$, assigned by the application layer, and each process must compute the disjunction of the input bits of all processes. DISJ is uniform, and works in an anonymous network under the distributed unfair daemon. The stabilization time of DISJ is $O(n)$ rounds, where n is the size of the network, and the memory requirement per process is $O(\log \mathcal{D} + \Delta)$ where $\mathcal{D}$ and Δ are, respectively, the diameter, and the maximum degree of the network.

keywords: anonymous network, disjunction, self-stabilization, silence, unfair daemon.

1 Introduction

We consider *self-stabilization* [2] in the *composite atomicity model* of computation [3]. A distributed algorithm is said to be *self-stabilizing* if, after any finite sequence of transient faults, it will eventually recover, without external (*e.g.*, human) intervention, provided that those faults do not corrupted its code. In particular, a self-stabilizing algorithm, starting from an arbitrary configuration, eventually behaves correctly.

A particular class of self-stabilizing algorithms is that of silent algorithms. A self-stabilizing algorithm is *silent* [4] if it converges to a global state where the values of the registers used by the algorithm remain fixed. In the composite atomicity model, a self-stabilizing algorithm is silent if and only if all its executions are finite. Silent self-stabilizing algorithms can be used to build fault-tolerant distributed data structures, such as spanning trees [5, 6, 7, 8] or clustering [9, 10, 11].

1.1 Contribution

In this paper, we study a new problem, which we call the *disjunction problem*. Given a network of processes, where each process x has a fixed *input bit*, $x.in$, the *disjunction problem* consists in computing $Output = \bigvee_{x \in V} x.in$, where V is the set of all processes of the network, and for each process to know the value of $Output$. In other words, each process should compute the disjunction of all input bits in the network.

In this paper we give a distributed solution to the disjunction problem, the algorithm DISJ, which is self-stabilizing and silent, and works under the *distributed unfair daemon*, the weakest scheduling assumption of the model.

DISJ is *uniform*, meaning that every process has the same program, and is *anonymous*, meaning that processes are not required to have distinguished IDs. Moreover, for each process x and each neighbor y of x, we assume that x has an area of its memory of size $O(1)$ dedicated to y, and that y can query this area specifically. The *stabilization time* of DISJ is $O(n)$ rounds, where n is the size of the network.

To evaluate the space complexity of DISJ, we introduce the *silent overflow* model, which allows to evaluate the minimal space requirement of an algorithm, provided that a mechanism preventing variable overflow is available. Such a mechanism simply consists in forbidding a process to execute an action that would cause its memory to overflow. In this model, the memory requirement can depend on global parameters like the number of processes n or the network diameter $\mathcal{D}$, although processes are not assumed to know these

*A preliminary version of this paper was presented at the international conference ICDCN'2013 [1].

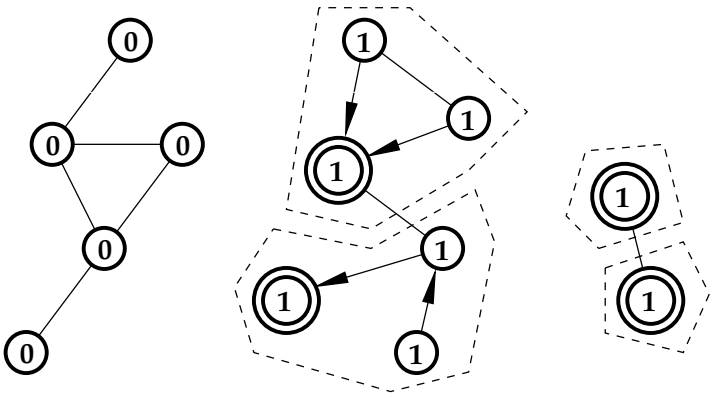


Figure 1: Examples of solutions computed by Algorithm DISJ: double circles represent nodes with input bit 1 (others have input bit 0); output bits are shown inside circles; arrows represents tree-edges.

parameters. In the silent overflow model, the space complexity of DISJ is $O(\log \mathcal{D} + \Delta)$ bits per process, where Δ is the degree of the network.

Additionally, when $Output = 1$, DISJ builds a breath-first search spanning forest of the network. Each process with input bit 1 is the root of its own tree. Each other process joins the tree rooted at the closest process whose input bit is 1, and each of those trees is a local BFS tree.

Examples of solutions computed by DISJ are given in Figure 1. In the leftmost network, there is no process with input 1; thus, all processes compute the output 0. In the rightmost network, all processes have input 1. So, they all compute output 1 and are all roots of their own trees. In the middle network, two processes have input 1. All processes have output 1. Moreover, two BFS trees are built.

1.2 Related Work

1.2.1 Shortest-Path Tree Construction with Disconnection Detection

Our proposal is close to the work of Glacet *et al* [12][1]. In [12], the authors give a silent self-stabilizing algorithm — in the composite atomicity model assuming a distributed unfair daemon — to maintain a shortest-path rooted tree and detect disconnected components. Roughly speaking, they consider a so-called *rooted* network, where there is a distinguished process, called *root*. Due to the dynamic nature of the network, the topology may become disconnected; in that case, starting from an arbitrary configuration, their algorithm converges to a terminal configuration where a shortest-path tree spans the component containing the root and where all processes in other components are in a state which indicates that they are isolated, meaning that they do not belong to the root component. Whenever all edges have weight one, a shortest-path tree is a BFS tree and so the problem solved by their algorithm can be seen as a particular case of our work, where only one process (the root) has input 1. Indeed, in this case, our algorithm also builds a BFS tree which spans the component of the root; all processes in other components stabilize to the output 0, meaning that they are isolated. Notice that complexities of the Glacet *et al.* algorithm are of the same order of magnitude as our algorithm DISJ.

1.2.2 The Count-To-Infinity Problem

The disjunction problem and the problem solved in [12] are closely related to preventing the *count-to-infinity* problem [13] in distance-vector routing protocols, where the distances to some unreachable process keep growing in routing tables because no process is able to detect the issue.

[1]Notice that this latter has been published in SSS'2014, while an earlier version of this paper appears in ICDCN'2013 [1].

1.2.3 BFS Construction and Leader Election

When exactly one process has input 1, our algorithm consists of a self-stabilizing BFS spanning tree construction. However, when all processes have input 0, the network is fully anonymous and BFS spanning tree constructions fail to solve the disjunction problem, as they face the *count-to-infinity* problem. Recall also that, contrary to the disjunction problem, the BFS spanning tree problem is impossible to solve in anonymous networks since leader election is impossible too [14]. However, notice that some techniques appearing first in self-stabilizing BFS spanning constructions [15] and self-stabilizing leader elections [16, 17] are reused in the algorithm we propose here.

Many self-stabilizing solutions for the BFS spanning tree construction have been proposed for arbitrary connected rooted networks [5, 18, 19, 20, 15]. In 1991, Chen *et al.* [5] presented the first self-stabilizing BFS tree construction in the composite atomicity model, but assuming a central daemon. In 1992, Huang and Chen [18] proposed the first self-stabilizing BFS tree construction in the composite atomicity model working under the distributed unfair daemon, its stabilization time is $\Theta(n)$ rounds in the worst case [21]. These two algorithms [5, 18] require that processes know the exact number of processes in the network. The algorithm given in [19] is not silent and has a stabilization time in $O(n + \mathcal{D}^2)$ rounds. The silent algorithm given in [20] has a stabilization time $O(\mathcal{D}^2)$ rounds. The Dolev *et al*'s algorithm [15] is written in the Read/Write atomicity model. This model is more general than the composite atomicity model. The algorithm does not assume any knowledge on any global parameter of the network, such as n for example. The counterpart being that there is no bound on process local memories. The algorithm is proven under the central fair assumption. However, its straightforward composite atomicity version has been proven to stabilize in $\mathcal{D}$ rounds in the worst case [21], assuming a distributed unfair daemon.

In connected identified networks, spanning tree constructions and Leader Election algorithms are closely related. Indeed, any BFS spanning tree construction requires a leader to be its root, and most of the silent self-stabilizing leader election algorithms also builds a spanning tree that is rooted at the leader node. Self-stabilizing algorithms for arbitrary connected identified networks that both elect a leader and build a spanning tree have been proposed in the message-passing model [17, 22, 7]. First, the algorithm of Afek and Bremler [17] stabilizes in $O(n)$ rounds using $\Theta(\log n)$ bits per process. 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, have been proposed in [22, 7]. 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, still for arbitrary connected identified networks, are also given in the composite atomicity model [23, 24, 25, 26, 27, 8, 28]. The algorithm proposed by Dolev and Herman [23] is not silent, works under a *fair* daemon, and assume that all processes know a bound N on the number of processes n. This solution stabilizes in $O(\mathcal{D})$ rounds using $\Theta(N \log N)$ bits per process. The algorithm of Arora and Gouda [24] works under a *weakly fair* daemon and assume the knowledge of some bound N on the number of processes n. This solution stabilizes in $O(N)$ rounds using $\Theta(\log N)$ bits per process. Datta *et al* [25] propose the first self-stabilizing leader election algorithm proven under the distributed unfair daemon. This algorithm stabilizes in $O(n)$ rounds. However, the space complexity of this algorithm is unbounded. Solutions in [26, 27, 8, 28] have a memory requirement which is asymptotically optimal, *i.e.*, in $\Theta(\log n)$ (the proof of optimality is given in [4]). The algorithm proposed by Kravchik and Kutten [8] assumes a synchronous daemon and the stabilization time of this latter is in $O(\mathcal{D})$ rounds. The solutions proposed in [26, 27, 28] assume a distributed unfair daemon and have a stabilization time in $O(n)$ rounds.

1.2.4 k-Dominating Sets and k-Clustering

Our algorithm DISJ finds application in network clustering. Indeed, it can be composed with a self-stabilizing *k-dominating set* algorithm to build a *k-clustering* of a network. In *k-dominating set* and *k-clustering* problems, k is a non-negative integer parameter which is given as constant input to all processes. Hence, in these two problems, k is *a priori* known by all processes. A k-clustering of the network is a distributed data structure which partitions processes into distinct subsets, called *clusters*, in which a process is distinguished as *clusterhead* and such that any member of a cluster is within distance k from its clusterhead. Usually,

every cluster is organized as a tree rooted at its clusterhead. The *k-dominating set* problem is related to the k-clustering problem as it allows designation of the set of clusterheads: a k-dominating set is a subset of processes S such that every process is within k hops of some member of S. Then, a k-clustering is obtained by having every process choose the closest member of S as its clusterhead. This latter task can be solved by our algorithm DISJ. Indeed, let $\mathcal{DS}$ be any self-stabilizing (silent) k-dominating set algorithm, *e.g.*, [29, 30, 31]. The output of $\mathcal{DS}$ is a bit at each process which states whether the process is in the dominating set, *i.e.,* whether it is a clusterhead. If we compose DISJ with $\mathcal{DS}$ using, for example, the hierarchical collateral composition [29] and considering the output of $\mathcal{DS}$ as the input bit in DISJ, then once $\mathcal{DS}$ has stabilized, DISJ will cause each non-member of the dominating set (with input bit 0) to join the tree rooted at the closest process with input 1 (*i.e.*, the tree of the closest clusterhead). Hence, we eventually obtain a spanning forest where each tree is a cluster of radius at most k, *i.e.*, a clustering of the network. Hence, we can obtain a self-stabilizing (silent) k-clustering algorithm simply by composing any self-stabilizing (silent) k-dominating set algorithm with our algorithm DISJ. Notice that self-stabilizing algorithms, written in the composite atomicity model, that directly build a k-clustering in an arbitrary connected identified network are given in [32, 9, 33, 10]. The solution in [9] stabilizes in $O(k)$ rounds using $O(k \log n)$ space per process. The algorithm given in [32] stabilizes in $O(k\,n)$ rounds using $O(k \log n)$ space per process. The algorithms given in [33, 10] stabilizes in $O(n)$ rounds using $O(\log k + \log n)$ space per process. Solutions in [32, 9, 33] assume a distributed unfair daemon, while the algorithm in [10] is proven under a distributed weakly fair daemon.

1.3 Outline of the Paper

In Section 2, we explain our model of computation. In Section 3, we first present an overview of DISJ, and then give the formal definition. In Section 4, we prove that DISJ is self-stabilizing and silent, and in Section 5 we show that DISJ stabilizes within $O(n)$ rounds. We give concluding remarks in Section 6.

2 Preliminaries

2.1 Distributed systems

We consider *distributed systems* of anonymous processes. Each process can communicate with a subset of other processes, called its *neighbors*. Communication is assumed to be bidirectional. Hence, the topology of the system can be represented as a simple undirected graph $G = (V, E)$, where V is the set of processes and E the set of edges, *i.e.,* communication links. We assume G is connected. Every process x can distinguish its neighbors using a *local labeling*. All labels of x's neighbors are stored into the set $N(x)$.

2.2 Composite Atomicity Model

We use the *composite atomicity model of computation* [2, 3] in which the processes communicate using a finite number of locally shared registers, called *variables*. Each process can read its own variables and those of its neighbors, but can write only to its own variables. The *state* of a process is the vector of values of all its variables. A configuration of the system is the vector of states of all processes. We denote by $\mathcal{C}$ the set of all possible configurations.

A distributed *algorithm* consists of one *program* per process. A distributed algorithm is called *uniform* if every process has the same program. In the present paper, all algorithms are uniform.

The program of a process x is a finite set of *actions* of the following form:

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

The *labels* are used to identify actions. The *guard* of an action in the program of process x is a Boolean expression involving the variables of x and its neighbors. If the guard of some action evaluates to true, then the action is said to be *enabled* at x. By extension, x is said to be enabled if at least one action is enabled at x, We denote by $\mathit{Enabled}(\gamma)$ the set of processes enabled in configuration γ. The *statement* of an action

is a sequence of assignments of the variables of x. An action can be executed only if it is enabled. In this case, the execution of the action consists of executing its statement.

The asynchronism of the system is represented by an adversary, called the *daemon*. In a configuration γ, if $\mathit{Enabled}(\gamma) \neq \emptyset$, then the daemon selects a non-empty subset S of $\mathit{Enabled}(\gamma)$ to perform an *(atomic) step*: each $x \in S$ atomically executes one of its actions enabled in γ, leading the system to a new configuration γ'. We write $\gamma \mapsto \gamma'$ if γ' can be reached from γ in one step. An *execution* is a *maximal* sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite or ends at a *terminal* configuration, *i.e.,* a configuration at which no process is enabled.

A *daemon* is usually defined in terms of *distribution* and *fairness*. Concerning the *distribution*, we assume here that the daemon is *distributed* meaning that, at each step, if one or more processes are enabled, then the daemon chooses at least one (possibly more) of these processes to execute an action. There are many different fairness refinements. For example, a daemon can be *weakly fair*, meaning that it eventually chooses every continuously enabled process. The *unfair* daemon is the weakest scheduling assumption: it can forever prevent a process from executing an action unless it is the only enabled process. In this paper, we assume that the daemon is *unfair*.

2.3 Rounds

To measure the time complexity of an algorithm, we use the notion of a *round* [34]. If a process x 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 say that x is *neutralized* during the step $\gamma_i \mapsto \gamma_{i+1}$. 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, and let $\mathcal{E}$ be the set of all possible executions of $\mathcal{A}$. A *specification* $\mathbb{S}$ is a predicate over $\mathcal{E}$.

$\mathcal{A}$ is *self-stabilizing* for $\mathbb{S}$ 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 $\mathbb{S}$.

The *stabilization time* is the maximum time (in steps or rounds) to reach a legitimate configuration starting from an arbitrary configuration.

2.5 Silent Overflow of Memory

It is common practice to assume that a given distributed algorithm $\mathcal{A}$ does not "know" either the size, n, or the diameter, $\mathcal{D}$, of the network. In fact, we make that assumption in this paper. On the other hand, it is common to claim that $\mathcal{A}$ has a given space complexity per process, and also that each process x has an integer variable, say $x.var$, which has no given upper bound. How can these assumptions be reconciled?

We use what we call the *silent overflow* model. Each process x has a capacity, say $B(x)$ bits, but no neighbor of x knows the value of $B(x)$. If there is some action, say Act, such that x is enabled to execute Act, but that action would cause the memory of x to overflow, and the daemon selects x to execute Act, then x appears, to its neighbors, to not have been selected. In that case, we say that x is *blocked*. However, x could be selected at a later step if, at that step, that selection causes Act to be neutralized. Alternatively, x becomes unblocked if changes of the neighbors of x allow x to execute Act without overflowing its memory. In that case, x simply executes Act.

There is no "signal" readable to $\mathcal{A}$ that a process x is blocked, meaning no guard of any action may use that information. If x is blocked, then it appears to any neighbor of x simply that x is not selected.

We now define space complexity in our model. If C is a function on processes, we say that $\mathcal{A}$ has space complexity $C(x)$ if, with the silent overflow rule $\mathcal{A}$ executes correctly provided $B(x) \geq C(x)$; and if we say that the space complexity of each process x is $O(f(x))$ for some function f, we mean that $C(x) = O(f(x))$.

2.6 Silence

In the composite atomicity model, an algorithm $\mathcal{A}$ is *silent* if and only if all its executions are finite [4]. Let γ be a terminal configuration. The set of all possible executions starting from γ is the singleton $\{\gamma\}$. Thus, if $\mathcal{A}$ is self-stabilizing and silent, γ must be legitimate. Thus, to prove that an algorithm is both self-stabilizing and silent for the disjunction problem, it is necessary and sufficient to show that all its execution are finite, and that there is an output variable $x.out$ at every process x such that $x.out = \bigvee_{x \in V} x.in$ in every terminal configuration. Notice that, in this case, the stabilization time is the maximum time (in steps or rounds) to reach a terminal configuration starting from an arbitrary configuration.

A configuration at which some process is enabled but all enabled processes are blocked is not a terminal configuration. Rather, at that configuration, $\mathcal{A}$ is *deadlocked* for lack of space.

3 Algorithm DISJ

We now present our algorithm, DISJ. Recall that the network is connected. DISJ computes *Output*, the disjunction of all input bits, and assigns *Output* to $x.out$ for each process x.

Moreover, if *Output* = 1, each process with input bit 1 becomes the root of a local BFS tree, and each process with input bit 0 belongs to the tree rooted at the nearest process with input bit 1. Ties are broken arbitrarily.

3.1 Overview of our Solution: Working Towards DISJ

We present a sequence of algorithms leading to DISJ.

1. DISJ_1 is a non-self-stabilizing asynchronous algorithm which solves the disjunction problem in $O(\mathcal{D})$ rounds. (DISJ_1 implements a principle similar to the BFS spanning tree construction given in [15].)

2. DISJ_2 is a modification of DISJ_1, which introduces the *resetting* of *invalid* processes (inspired by [35]). DISJ_2 always works if *Output* = 1, but sometimes fails if *Output* = 0, because it may not be able to get rid of *false trees,* sets of processes which appear to be fragments of BFS trees. We show an example of how DISJ_2 can livelock.

3. DISJ_3. By introducing *color waves*, first defined in [16], we retard the growth of false trees, permitting the reset action to eliminate them if *Output* = 0. As a side effect, the computation in the case that *Output* = 1 is made more difficult. Although DISJ_3 has round complexity $O(n)$, it might never terminate under the distributed unfair daemon; the last round might be infinite since an enabled process might never be selected.

4. DISJ_4 = DISJ. In order to force termination of our final algorithm, we introduce a feature which eliminates the possibility of endless color waves in a completed BFS tree.

3.1.1 The Non Self-Stabilizing Algorithm DISJ_1

We now define our simple non-self-stabilizing algorithm, DISJ_1. Recall that $x.in$ is the input bit of a process x, and that *Output* is the disjunction of all input bits. Write $I_i = \{x : x.in = i\}$, for $i = 0, 1$. Thus,

Output $= 0$ if $I_1 = \emptyset$, and *Output* $= 1$ otherwise. If x, y are processes, let $||x, y||$ be the distance ("hop-distance") from x to y. If S is a non-empty set of processes, let $||S, y|| = \min\{||x, y|| : x \in S\}$. We let $||\emptyset, y|| = \infty$. Finally, let $L(x) = ||I_1, x||$.

In DISJ$_1$, each process x has the following four variables.

$x.in \in \{0, 1\}$**:** the *input bit* of x.

The input bits are given by the application, and not changeable by the algorithm.

$x.out \in \{0, 1\}$**:** the *output bit* of x.

When the execution halts, $x.out = Output$. We let $O_i = \{y : y.out = i\}$, for $i = 0, 1$. During an execution the sets O_i may change, but eventually, one will contain all processes and the other will be empty.

$x.par \in N(x) \cup \{\perp\}$**:** the *parent* of x.

If *Output* $= 0$, then $x.par = \perp$ when the execution halts. If *Output* $= 1$, then, when the execution halts, $x.par = \perp$ for all $x \in I_1$, while $x.par$ is the parent pointer of x in the local BFS tree rooted at the nearest member of I_1 if $x \in I_0$.

We assume that for every $y \in N(x)$, x has a single bit register dedicated to y, readable by y. This bit is 1 if $y = x.par$, and 0 otherwise. Hence, y is able to evaluate whether $y = x.par$ in the guard of its actions.

$x.lvl \geq 0$**:** integer or ∞, the *level* of x.

When the execution halts, $x.lvl = L(x)$ (in particular, if *Output* $= 0$, $x.lvl = \infty$).

The following explanation is illustrated with Figures 2 and 3. In these figures, double (resp. simple) circles represent process with input 1 (resp. 0). Numbers inside circles represent outputs. Numbers next to circles represent level values.

Initially, all processes are assigned to the *zero state*: in this state, $x.out = 0$, $x.par = \perp$, and $x.lvl = \infty$. Then, DISJ$_1$ consists of two actions:

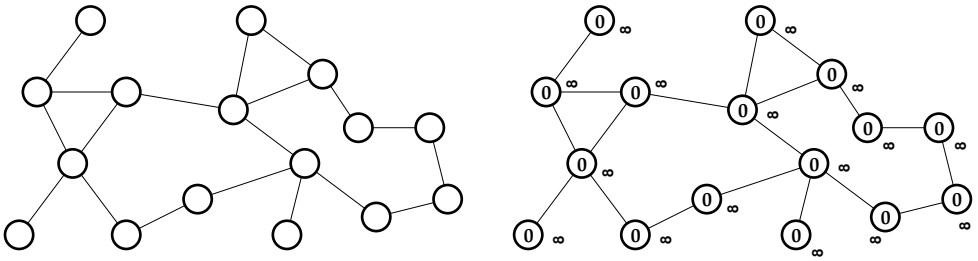
$$\begin{array}{lcll} \text{Init} & :: & x.in = 1 \wedge x.out = 0 & \rightarrow \quad x.out = 1 \\ & & & \quad\;\; x.lvl = 0 \\ \\ \text{Join } y & :: & x.in = 0 \wedge x.lvl \neq Level(x) & \rightarrow \quad x.out = 1 \\ & & \wedge y.lvl = Level(x) - 1 & \quad\;\; x.par = y \\ & & & \quad\;\; x.lvl = Level(x) \end{array}$$

where $Level(x) = 1 + \min\{y.lvl : y \in N(x)\}$.

In the case where all input bits are zero (see Figure 2(a)), nothing happens after initialization (see Figure 2(b)): the initial configuration is also terminal, and all output bits are correct.

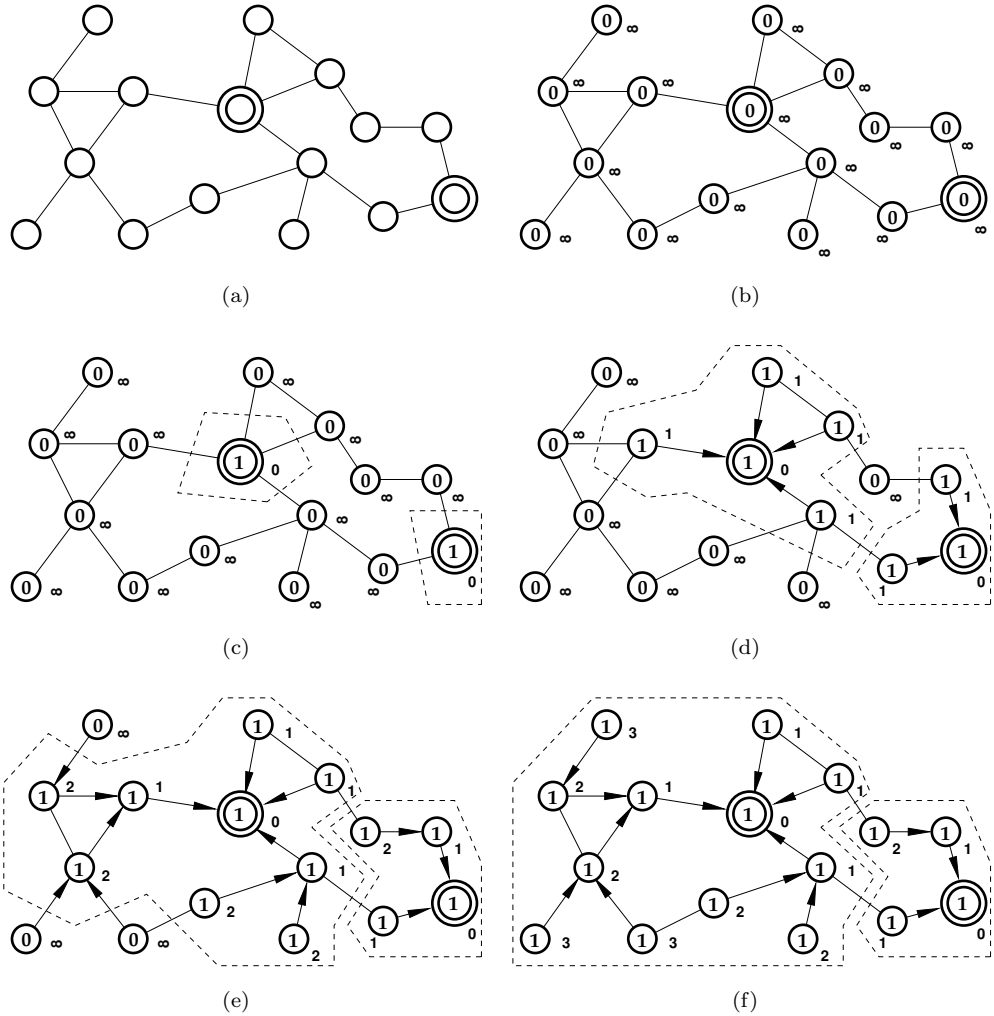
Consider now the case where some inputs are 1 (see Figure 3(a)). After initialization (see Figure 3(b)), every process x such that $x.in = 1$ changes its output, $x.out$, to 1 and its level, $x.lvl$, to 0, becoming a BFS tree root, see Figure 3(c). In the next round, $y.out$ changes to 1 and $y.lvl$ changes to 1 for any y which is a neighbor of a member of I_1, and $y.par$ changes to point to a neighboring member of I_1. (We say that a process x recruits a process y if y executes an action to choose x as parent. In this case, we also say that y joins x.)

Continuing, if $L(x) = \ell$ for $\ell > 1$, then within $\ell + 1$ rounds, $x.out \leftarrow 1$, $x.lvl \leftarrow L(x)$, and $x.par \leftarrow y$, where y is an arbitrary neighbor of x such that $L(y) = \ell - 1$. See, *e.g.*, Figures 3(d) and 3(e). After at most $\mathcal{D} + 1$ rounds, the network reaches a terminal configuration, where $\mathcal{D}$ is the diameter of the network (see Figure 3(f)).



(a) (b)

Figure 2: The non-self-stabilizing algorithm DISJ_1 when all inputs are zero.



(a) (b)

(c) (d)

(e) (f)

Figure 3: Execution of DISJ_1 when some inputs are 1.

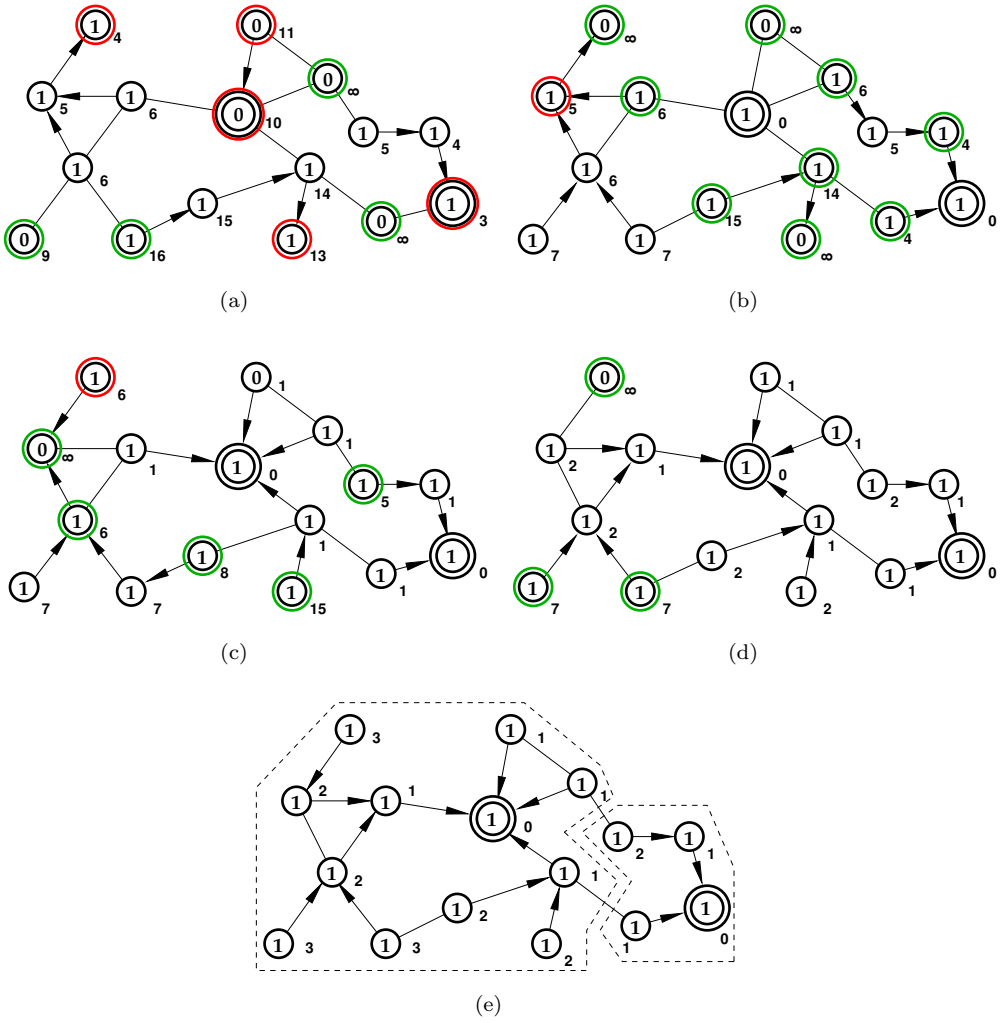


Figure 4: Execution of DISJ_2 when some inputs are 1: nodes in red represent processes enabled to reset; nodes in green represent processes that are enabled to join. because of the previous rule.

3.1.2 Local Resets: DISJ_2

DISJ_1 is not self-stabilizing: given an arbitrary configuration, it may not converge to a solution if $Output = 0$. We now consider DISJ_2, which is simply DISJ_1 with one additional action, *Reset*.

In DISJ_2, we define a process x to be *valid* if one of the following three conditions holds:

1. $x.in = x.out = 1$, $x.lvl = 0$, and $x.par = \perp$,
2. $x.in = 0$, $x.out = 1$, $x.lvl = Level(x)$, and $x.lvl = 1+x.par.lvl$, where $Level(x) = 1+\min\{y.lvl : y \in N(x)\}$,
3. x is in the zero state, *i.e.,* $x.out = 0$, $x.par = \perp$, and $x.lvl = \infty$.

Any process which is not valid is said to be *invalid*.

In DISJ_2, a process is enabled to execute a reset action if it is invalid. This action changes its state to the zero state.

DISJ_2 converges to a legitimate configuration within $O(\mathcal{D})$ rounds, provided $Output = 1$. We illustrate an example execution in Figure 4.

First, assume that γ contains some input bits equal to 1, *e.g.*, in Figure 4(a). Consider any process x such that $x.in = 1$. If $x.lvl \neq 0$ or $x.out \neq 1$, the state of x is invalid and it resets, as shown in Figure 4(b). As in DISJ_1, by dynamic programming, the system reaches a terminal configuration in $\mathcal{D}+1$ rounds, as shown in Figure 4(e).

The case where $Output = 0$ is more difficult. In fact, DISJ_2 may enter a livelock. If not all processes are in the zero state, the configuration will contain at least one false tree, and each false tree must contain an invalid process at its root. Invalid processes will eventually reset top-down in the false trees; the problem is that a false tree can continue to recruit at its leaf end, and the cycle might never end.

We give an example of such a livelock in Figure 5, processes with levels 3 to 6 are arranged in a chain. We assume that each enabled process executes at each step. While the process at level 3 resets, another process joins the chain at level 7, and so forth. After six steps, the system reaches the configuration shown in 5(g), which is similar to the configuration 5(a), except that the levels have increased by six. As processes do not know any bound on the diameter or the size of the network, the execution continues, theoretically forever.

3.1.3 Color Waves: DISJ_3

We prevent livelock by the introduction of *color waves*, first given in [16]. We have two kinds of trees: *false* trees, *i.e.*, sets of processes which appear to be fragments of BFS trees, and *true* trees, *i.e.*, sets of valid processes organized as trees rooted at valid members of I_1. Of course, we do not have true trees if $Output = 0$. Color waves ensure that false trees eventually stop growing at their leaf end so that they eventually disappear by propagation of reset from their root end. Moreover, color waves do not prevent true trees from growing.

We give a more extensive intuitive explanation of color waves in Section 3.5.

To implement color waves, DISJ_3 requires an additional variable:

$x.color \in \{0,1\}$**:** the *color* of x.

True and False Trees From now on we refer to any invalid process x such that $x \in O_1$ as a *false root*. Then, we split valid processes into two categories: *true roots* and *true children*. A true root is any valid process x such that $x.in = 1$. Any valid process y which is not a true root satisfies, in particular, $y.par \neq \perp$. Then, y is said to be a *true child* of the process designated by $y.par$. Such a process y is linked to some root x (either a true, or a false one) by a path $x_0 = y, x_1, \ldots, x_k = x$ such that $\forall i \in [0..k-1]$, x_i is a true child of x_{i+1}. Such a path is acyclic. (We will see in Section 3 that the true child relation implies that $x_i.lvl \neq x_{i+1}.lvl + 1$, $\forall i \in [0..k-1]$.) So, given a true root (resp. the false root) x, we define the *true (BFS) tree* (resp. *false (BFS) tree*) rooted at x as the set of processes linked to x by such a path.

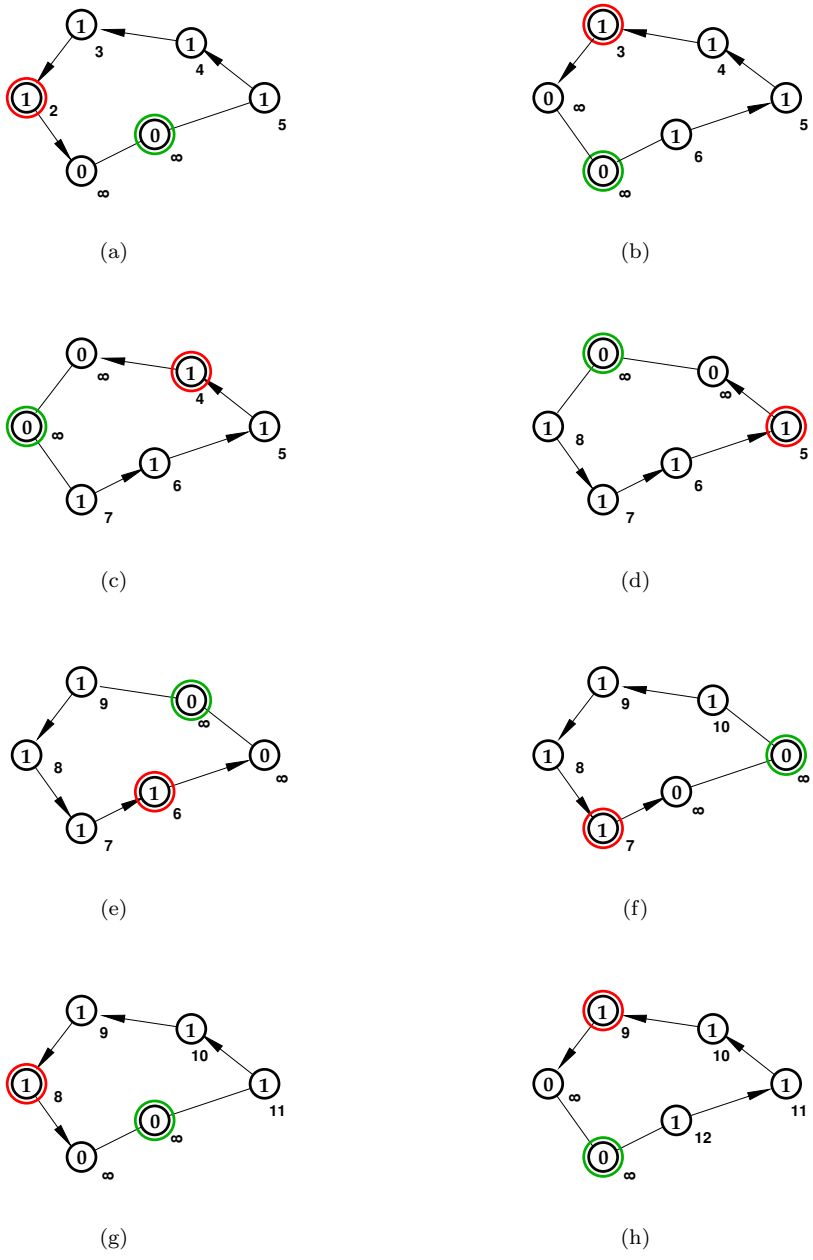


Figure 5: Execution of DISJ_2, when all inputs are zero: nodes in red represent processes enabled to reset; nodes in green represent processes that are enabled to join.

Color Waves in True Trees Waves of alternating colors move from the leaves of a growing true tree to its true root which absorbs the waves. A process x can join a process y only if $y.color = 1$, and after joining, $x.color = 0$. Thus x cannot recruit until its 0-wave has reached $y = x.par$ and x has also initiated a new 1-wave. Thus each layer of growth of the tree causes two color waves to move up. Color waves are not allowed to pass each other.

More precisely:

1. For every process x, x can join y (by, in particular, setting $x.par$ to y), only if x is in the zero state (*i.e.*, x does not belong to any tree) and $y.color = 1$. Moreover, if x makes that choice, it sets, in particular, $x.color$ to 0.

2. A process x can change its color, $x.color$, only if it satisfies the following three conditions:

   (a) x is either a true root, or a true child and $x.par.color = x.color$.

   (b) All true children of x (if any) have the opposite color from x.

   (c) Either $x.color = 0$ or $Recruits(x) = \emptyset$. $Recruits(x)$ is the set of neighbors of x that might join x in the future, as defined in Section 3.2.

   Let y be any neighbor of x. Roughly speaking, y might join x in the future if $y.out = 0$ or $y.lvl > x.lvl + 1$. In the former case, y should join a BFS tree maybe by joining x. In the latter case, y should eventually select a new parent, maybe x, to reduce its level.

An example execution of DISJ_3 is given in Figure 6, where the rightmost process x is the sole member of I_1. In the first step, x first initializes to become a true root, and can then recruit its neighbor y. When y selects x as parent, y takes color 0. Then, process x absorbs the 0-wave, after which y can start a new 1-wave, and so can recruit the next process to its left, and so forth.

Color Waves in False Trees It is necessary to delete all such false trees. This is an easy task if $Output = 1$, but is difficult if $Output = 0$. If a process x is invalid, it will delete itself simply by resetting. In particular, if x was a false root, then all its true children (if any) become false roots. Thanks to color waves, two rounds are required for the height of a local BFS tree to increment, while at most one round is necessary for a false root to reset. In addition, color waves ensure that each false tree eventually stops growing at the leaf end while it continues to delete itself at root end.

Recall that a process can recruit a new process only if its color is 1, and the recruited process will initially have color 0. As the color of the color waves in a tree must alternate, color waves must move upward to allow a new color wave 1 to appear at leaves and so to allow new nodes to be recruited. Colors in a tree cannot pass each other. To allow the color waves to continue upward, the root of the tree must "absorb" each wave. Only a *true root* is allowed to absorb a wave, false roots cannot. Hence, color waves will not be absorbed in false trees, and, in the worst case, each false tree will eventually be *color-locked*, meaning that there can be no further color changes in that tree. In particular, if a tree is color locked, then $x.color \neq x.par.color$ for any true child in the tree. In such a case, the waves are maximally crowded and cannot move up in the tree. In DISJ_3, because of the phenomenon of color locking, a false tree cannot continue recruiting processes forever. The processes of the false tree reset top-down, eliminating the tree.

Thanks to resets and color waves, the system reaches a legitimate configuration where all processes have correct output and a spanning forest is defined (if there at least one input bit equal to 1) within $O(n)$ rounds, provided the daemon is weakly fair. Under the distributed unfair daemon, DISJ_3 might never terminate, as a true BFS tree can generate color waves at the leaves and absorb them at the root forever. This cycle could occur even if not all local BFS trees are correct. This drawback is corrected in DISJ_4 below.

3.1.4 Silence: $\text{DISJ}_4 = \text{DISJ}$

To obtain silence, we use an additional boolean variable:

$x.done$**:** Boolean.

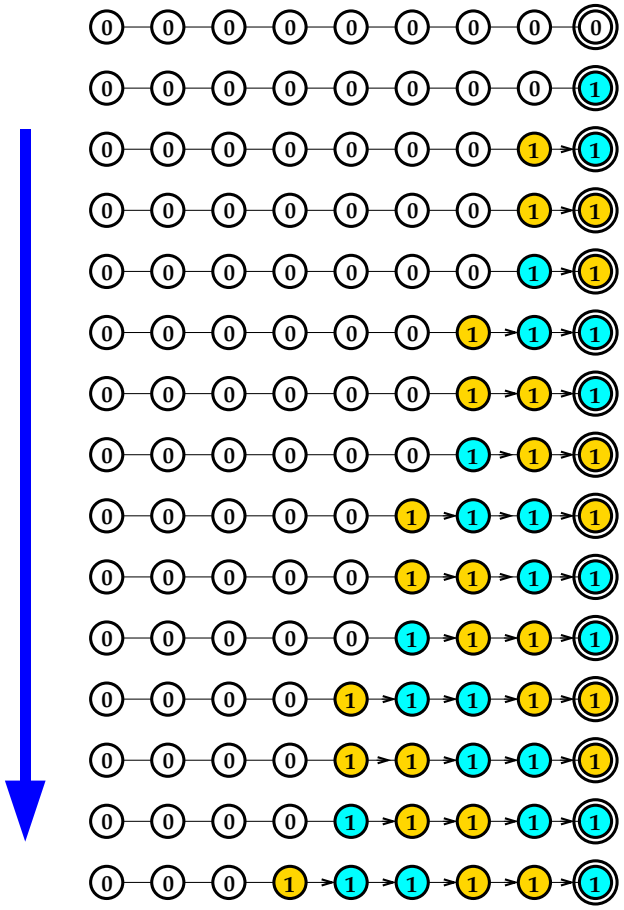


Figure 6: Example of color waves in a chain network. The doubly circled process is the sole member of I_1. The internal bit is the value of $x.out$. Color 0 for members of O_1 is represented by gold. Color 1 for members of O_1 is represented by cyan. Colors are irrelevant for members of O_0, so these processes are colored white. When an arrow is outgoing from a process, it points to its parent. When no arrow is outgoing from a process x, this means that $x.par = \perp$. The blue arrow shows the flow of time.

This variable is irrelevant if $x.out = 0$. If $Output = 1$, and a local BFS tree is correct, all $x.done$ are changed to 1 in a convergecast wave. When $r.done = 1$ for $r \in I_1$, it stops absorbing color waves, and the tree becomes *color-locked* within $O(\mathcal{D})$ rounds. When $x.done = 1$ for all x, and all local trees are correct and color-locked, the execution ends.

More precisely:

1. If a leaf process x cannot recruit neighbors anymore, then $x.done$ is set to 1.
2. If a process x with true children cannot recruit neighbors anymore and $y.done$ holds for all its children y, then $x.done$ is set to 1.
3. If $x.done$ holds at a true root x, then x cannot change its color.

We now give the formal definition of Algorithm DISJ. In Subsection 3.2, we define the variables of DISJ, and then some functions and sets used by DISJ. We formally define *legitimacy* of DISJ in Subsection 3.3. Actions of DISJ are explained in Subsection 3.4.

3.2 Variables, Functions and Sets

We first list the variables for each process x.

1. $x.in \in \{0,1\}$, the *input bit* of x. This bit is given by the application layer, and cannot be changed by DISJ.
2. $x.out \in \{0,1\}$, the *output bit* of x.
3. $x.par \in N(x) \cup \{\perp\}$, the *parent* of x.
4. $x.lvl$, non-negative integer or ∞, the *level* of x.
5. $x.color \in \{0,1\}$, the *color* of x. This variable is taken from [16]. As in that paper, the purpose of the color variable is to limit the growth of *false trees,* as defined below.
6. $x.done$, Boolean.

For convenience, we define the following sets of processes.

7. $I_i = \{x : x.in = i\}$ and $O_i = \{x : x.out = i\}$, for $i \in 0, 1$.
8. $Chldrn(x) = \{y \in N(x) : (y.par = x)\}$, the *children* of x.

The following functions can be computed by any given process x by examining its own and its neighbors' variables.

9. $Level(x) = \begin{cases} 0 \text{ if } x \in I_1 \\ \infty \text{ if } N(x) \cap O_1 = \emptyset \\ 1 + \min\{y.lvl : y \in N(x) \cap O_1\} \text{ otherwise} \end{cases}$
10. $True\_Root(x) \equiv (x \in I_1 \cap O_1) \wedge (x.lvl = 0) \wedge (x.par = \perp)$, x is a *true root.*
11. $True\_Child(x) \equiv (x \in O_1) \wedge (x.par \in O_1) \wedge (x.lvl = Level(x) = 1 + x.par.lvl)$, Boolean, x is a *true child.*
12. $True\_Chldrn(x) = \{y \in Chldrn(x) : True\_Child(y)\}$, the *true children* of x.
13. $0\_Valid(x) \equiv (x.out = 0) \wedge (x.par = \perp) \wedge (x.lvl = \infty)$, Boolean, x is in a valid state with output bit 0.
14. $1\_Valid(x) \equiv True\_Root(x) \vee True\_Child(x)$, Boolean, x is in a valid state with output bit 1.
15. $Valid(x) \equiv 1\_Valid(x) \vee 0\_Valid(x)$, Boolean, x is *valid.* If $\neg Valid(x)$, we say x is *invalid.*
16. $Recruits(x) = \{y \in N(x) : (y \in O_0) \vee (y.lvl > 1 + x.lvl)\}$, the neighbors of x that might join x.
17. $Done(x) \equiv (x \in O_1) \wedge (Recruits(x) = \emptyset) \wedge (\forall y \in True\_Chldrn(x) : y.done)$, Boolean.

3.3 Legitimate Configurations

There are two kinds of legitimate configurations.

1. We say that a configuration is *legitimate of type 0* if $V = I_0 = O_0$, *i.e.,* the input and output bits of every process are zero.

2. We say that a configuration is *legitimate of type 1* if the following conditions hold.

   (a) $I_1 \neq \emptyset$,

   (b) $1\_Valid\,(x)$ for all x,

   (c) $x.par.color \neq x.color$ if x is any true child,

   (d) $x.done$ for all x.

3.4 Actions of DISJ

Priorities We list the actions of DISJ in Table 1. The first column of the table gives the name of the action, as well as its *priority*. Priorities are used to simplify the design of DISJ, as explained below. The second column gives an informal name of the action. The fourth column of Table 1 lists the *statement* of each action. Note that **reverse** $x.color$ in Actions A5 and A6 just consists in reversing the value of $x.color$ (0 becomes 1, and conversely).

The guard of each action is a Boolean function, which we express as a list of *clauses* in the third column. Each guard is the conjunction of the clauses. If the priority of an action is not 1, its guard has an additional unlisted clause; this clause states that no action of higher priority is enabled.

The Spanning Forest $\mathcal{F}$ For any configuration, let $\mathcal{F}$ be the directed graph whose nodes are all processes, and whose edges are all ordered pairs (x, y) such that $x \in True\_Chldrn\,(y)$, *i.e., True_Child* (x) and $x.par = y$. $\mathcal{F}$ is acyclic since $x.par.lvl < x.lvl$. Since the out-degree of $\mathcal{F}$ is bounded by 1, $\mathcal{F}$ must be a forest. For $x \in O_1$ we let T_x be the set of descendants of T. Recursively defined, T_x consists of x and T_y for all $y \in True\_Chldrn\,(x)$. $\mathcal{F}$ is the disjoint union of rooted trees T_r, where r is either a true root or a false root, *i.e.,* an invalid process in O_1. We call those trees *true trees* or *false trees*, accordingly.

Explanations of the Actions of DISJ We now give a detailed explanation of each of the actions of DISJ. We call Actions A1, A2, and A3, *structure actions*, because they change the topology of the directed graph $\mathcal{F}$.

Action A1 (Initialize): If x is a true root at the initial configuration, it will remain a true root for the remainder of the execution, and will never be enabled to execute any structure action. If $x.in = 1$ and x is not initially a true root, then x is enabled to execute Action A1, becoming a true root. Once x is a true root, it will never again execute any structure action.

Action A2 (Reset): It is mainly through this action that false trees are eventually eliminated. All invalid processes are enabled. If an invalid process $x \notin I_1$ is selected, it will execute Action A2, becoming 0-valid.

Action A3 (Join y): If a process $x \in O_0$ has a neighbor $y \in O_1$, where $y.color = 1$ and $y.lvl = Level(x)$, then x is enabled to join y by executing Action A3, provided x has no children. When x joins y, $x.color \leftarrow 0$. The purpose of the condition $Chldrn\,(x) = \emptyset$ is to prevent an execution of this action from causing more than one process to join the tree.

Action A4 (Finish): If $x \in O_1$, and if it appears to x, by looking at its own and its neighbors' variables, that construction of the local BFS trees is done, then $x.done$ is changed to 1. Alternatively, if $x \in O_1$

Table 1: Actions of DISJ

A1 priority 1	Initialize	$x.in = 1$ $\neg\, True\_Root\,(x)$	$\rightarrow$	$x.out \leftarrow 1$ $x.color \leftarrow 1$ $x.lvl \leftarrow 0$ $x.par \leftarrow \perp$ $x.done \leftarrow 0$
A2 priority 2	Reset	$x.in = 0$ $\neg\, Valid(x)$	$\rightarrow$	$x.out \leftarrow 0$ $x.lvl \leftarrow \infty$ $x.par \leftarrow \perp$
A3 priority 3	Join y	$x.in = 0$ $y \in N(x)$ $y.out = 1$ $y.color = 1$ $1 + y.lvl = Level\,(x)$ $x.out = 0$ $Chldrn\,(x) = \emptyset$	$\rightarrow$	$x.par \leftarrow y$ $x.color \leftarrow 0$ $x.out \leftarrow 1$ $x.lvl \leftarrow y.lvl + 1$ $x.done \leftarrow 0$
A4 priority 4	Finish	$x.out = 1$ $x.done \neq Done\,(x)$	$\rightarrow$	$x.done \leftarrow Done\,(x)$
A5 priority 5	Reverse Color	$True\_Child\,(x)$ $x.par.color = x.color$ $\forall y \in Chldrn\,(x) : y.color \neq x.color$ $(Recruits\,(x) = \emptyset) \vee (x.color = 0)$	$\rightarrow$	**reverse** $x.color$
A6 priority 5	Absorb Color Wave	$True\_Root\,(x)$ $\forall y \in Chldrn\,(x) : y.color \neq x.color$ $(Recruits\,(x) = \emptyset) \vee (x.color = 0)$ $\neg x.done$	$\rightarrow$	**reverse** $x.color$

can determine that the local BFS trees are not finished, $x.done$ is changed to 0. Both changes are accomplished by the execution of Action A4.

For $Output = 1$, when construction of local BFS trees is finished, all the *done* variables change to 1 in a convergecast wave beginning at the leaves. When that wave reaches a true root, that root stop absorbing colors, causing *color-lock* to percolate down its tree. When all trees are color-locked, the configuration is terminal.

Action A5 (Reverse Color): Color waves alternate in color, and no color wave can pass a preceding color wave. This rule is enforced by the guard of A5. In order for the next color wave to reach x, that wave must have already reached all children of x (if there are no children, then x initiates a new color wave by executing A5) and the current color wave of x must already have reached $x.par$.

Action A6 (Absorb Color): Since color waves alternate colors and cannot pass each other, eventually every chain would have alternating colors, *i.e.,* x and y would have different colors if $y = x.par$. This situation is called *color-lock*. A color-locked chain can only recruit a process if its last process has color 1, and after it recruits that new process, which then has color 0, no further recruitment is possible. Thus, in order for the local BFS trees to grow, it is necessary for the root processes to *absorb* color waves. Action A6 by a true root x consists of simply allowing the color wave that has reached its children to move up to x. This then destroys (absorbs) the process' current color wave.

3.5 Intuitive Explanation of Color Waves and Energy

The purpose of the color wave mechanism is to limit the growth of false trees. Color waves are implemented by the algorithm itself. Each color wave is initiated at the leaves and moves toward the root. Color waves alternate colors, and cannot pass each other. A true root can absorb color waves, while a false root cannot. Thus, a false tree will eventually become color-locked if it is not eliminated first.

We say that a tree T is color-locked if no process in T has the same color as its parent. In a color-locked tree, color waves cannot move.

The purpose of the notion of energy is to prove that false trees are eliminated. The algorithm itself does not implement energy; it is only a virtual quantity used during the proof.

Every tree T has a value of energy, $Energy(T)$, which is defined bottom-up. As T recruits new members at the leaves, $Energy(T)$ does not change. As the root of a true tree T absorbs a color wave, $Energy(T)$ may increase. As the root of a false tree resets, energy decreases. The maximum possible value of $Energy(T)$ is $2n$, and therefore all false trees will be eliminated within $2n$ rounds.

We have to prevent color waves from being generated forever in true trees. When a true tree has finished recruiting processes, the boolean variable done changes to true for all processes in a bottom-up wave. When $r.done$ is true for the root r of a true tree T, r stops absorbing color waves. Eventually T becomes color-locked.

If Output = 0, DISJ converges when there are no false trees. If Output = 1, DISJ converges when there are no false trees, and all true trees are color-locked.

4 Correctness of DISJ

We first prove partial correctness of DISJ, then that there is no infinite execution of DISJ. Combining these two results, we obtain correctness of DISJ, namely that the algorithm is self-stabilizing and silent.

4.1 Partial Correctness of DISJ

Partial correctness of DISJ means that if any execution of DISJ halts, the output is correct.

Lemma 4.1 *In a terminal configuration of* DISJ*, for any process* x

(a) x *is valid,*

(b) *if* $True\_Child(x)$*, then* $x.par.color \neq x.color$*,*

(c) *if* $x \in O_1$ *then* $Done(x) = x.done = 1$*.*

Proof. If x is not valid, then x is enabled to execute Action A2, contradiction. Therefore (a) holds.

Suppose (b) does not hold. Pick x of maximum level such that $x.par.color = x.color$. Then $y.color \neq x.color$ for all $y \in True\_Chldrn(x)$ (since $y.lvl = x.lvl + 1$, by definition of $True\_Chldrn(x)$). If $x.color = 1$ and $Recruits(x) \neq \emptyset$, then any member of $Recruits(x)$ is enabled to execute Action A3, contradiction. If $x.color = 0$ or $Recruits(x) = \emptyset$, then x is enabled to execute Action A5, contradiction. Thus (b) holds.

We prove (c) by induction of $x.lvl$. We first note that $x.done = Done(x)$, since otherwise x is enabled. Suppose $x.done = 0$. If $x.lvl = 0$, then x is a true root. If $x.color = 0$ or $Recruits(x) = \emptyset$, then x is enabled to execute Action A6, contradiction. Otherwise, any member of $Recruits(x)$ is enabled to execute because $x.color = 1$, contradiction.

Now suppose $x.lvl > 0$. By the inductive hypothesis, $x.par.done = 1$, and thus $Done(x.par) = 1$, since otherwise $x.par$ would be enabled. Hence $x.done = 1$. □

Corollary 4.2 *In a terminal configuration of* DISJ*,* $Recruits(x) = \emptyset$ *if* $x \in O_1$*.*

Proof. If $Recruits(x) \neq \emptyset$ then $Done(x) = 0$, contradicting Lemma 4.1. □

Corollary 4.3 *In a terminal configuration of* DISJ*, all output bits are the same.*

Proof. If not all output bits are the same, there is some $x \in O_1$ which has a neighbor in O_0, which implies that $Recruits(x) \neq \emptyset$, contradicting Corollary 4.2. □

Lemma 4.4 *Any terminal configuration of* DISJ *is legitimate.*

Proof. We study the two following cases:

- Assume that $Output = 0$. Then, by Lemma 4.1(a), all processes are valid. Assume, by contradiction, that there is a process x such that $1\_Valid(x)$ holds. Without loss of generality, assume that $x.lvl$ is minimum. Then, $1\_Valid(x)$ implies, in this case, that $x.par \in O_1$. Let $y = x.par$. $1\_Valid(y)$ also holds. Now, $1\_Valid(x)$ implies that $y.lvl = x.lvl - 1$, which contradicts the fact that $x.lvl$ is minimum. Hence, $0\_Valid(p)$ holds for every process p, which implies that the configuration is legitimate (of type 0).
- Assume that $Output = 1$. Let x such that $x \in I_1$. As, Action A1 is disabled at x, $True\_Root(x)$ holds, which in particular means that $x \in O_1$. So, all processes are in O_1 by Corollary 4.3. Finally, by Lemma 4.1, we are done: the configuration is legitimate (of type 1).

□

4.2 No Infinite Execution

We now know that DISJ is partially correct, but we need to prove that DISJ converges, instead of possibly entering livelock.

We use the method of color waves to prove that DISJ converges if $Output = 0$. However, the same color waves that are useful if $Output = 0$ make the proof of convergence harder in the case that $Output = 1$.

Any infinite execution of DISJ must contain a cycle, as we show in Lemma 4.9, and there is no cycle, as we show in Lemma 4.11. We conclude, in Theorem 4.12, that DISJ is correct.

We borrow the concept of *energy* from [16] to handle this problem. At any configuration of DISJ, and for any process x, let

$$Energy(x) = \begin{cases} 0 \textbf{ if } x \in O_0 \\ 1 \textbf{ if } (x \in O_1) \wedge (True\_Chldrn(x) = \emptyset) \wedge (x.color = 0) \\ 2 \textbf{ if } (x \in O_1) \wedge (True\_Chldrn(x) = \emptyset) \wedge (x.color = 1) \\ \max \Big\{ \{1 + Energy(y) : (y \in True\_Chldrn(x)) \wedge (y.color \neq x.color)\} \\ \quad \cup \{2 + Energy(y) : (y \in True\_Chldrn(x)) \wedge (y.color = x.color)\} \\ \quad \textbf{otherwise} \Big\} \end{cases}$$

We define *Max_Energy* to be the maximum energy of process in the network.

Lemma 4.5 *Max_Energy* $\leq 2n$.

Proof. If $x \in O_0$, then $Energy(x) = 0$. By the definition of energy, and by induction, if $x \in O_1$, then $Energy(x) \leq 2d + 2 \leq 2n$, where d is the length of the longest directed path in $\mathcal{F}$ from a leaf x. □

Lemma 4.6 *If $x \notin I_1$, an action of x does not cause Max_Energy to increase.*

Proof. Suppose x executes an action at step s. We consider the effect of that action on *Max_Energy*. We first note that x cannot execute Actions A1 or A6 since $x \notin I_1$.

Case 1: x executes Action A3. Let $y = x.par$ after step s. $Energy(y) \geq 2$ before step s, while $Energy(x) = 1$ and $y.color \neq x.color$ after step s. Thus, *Max_Energy* does not increase.

Case 2: x executes Action A2. Then $Energy(x) = 0$ after the step. The energy of any ancestor of x might decrease as a result of this action, but no value of *Energy* can be caused to increase.

Case 3: x executes Action A4. This action has no effect on any value of *Energy*.

Case 4: x executes Action A5. Since $x \notin I_1$, x is a true child. Before the step, $x.par$ has the same color as x, while all children of x have the opposite color. Let $E = Energy(x)$ before step s. Then before step s, $Energy(x.par) \geq E+2$ and $Energy(y) \leq E-1$ for all $y \in True\_Chldrn(x)$. After the step, $Energy(x) = E+1$ but x and $x.par$ have now opposite color, so no other values of *Energy* are changed. In particular, we already had $Energy(x.par) \geq E + 2$. Hence *Max_Energy* does not increase.

In all cases, *Max_Energy* does not increase. □

Let $\xi(x) = \begin{cases} 0 \text{ if } x \in O_0 \\ x.lvl + Energy(x) \text{ if } x \in O_1 \end{cases}$

Lemma 4.7 *If x is a true child, then $\xi(x.par) \geq \xi(x)$.*

Proof. $Energy(x.par) \geq 1 + Energy(x)$. □

Write $\Xi = \max \begin{cases} 2n \\ \max\{\xi(x)\} \end{cases}$

Lemma 4.8 Ξ *does not increase during an execution of* DISJ.

Proof. We consider the effect on Ξ of one execution of some process x.

Case 1. If x executes Action A1, then $\xi(x) \leftarrow 2$, and $\xi(y)$ remains unchanged for any $y \neq x$.

Case 2. Suppose x executes Action A3 at step s, and $x.par \leftarrow y$. Let $\ell = y.lvl$ and $E = Energy(y)$ before step s. After step s, $Energy(x) = 1$ and $x.lvl = \ell + 1$. Thus $\xi(x)$ after the step is not greater than $\xi(y)$ before the step, since $E \geq 2$.

Case 3. Suppose x executes Action A2 at step s. Then $\xi(x) \leftarrow 0$. The effect of this action on other processes is only to decrease the energy of some processes, and so no value of ξ can increase.

Case 4. Suppose x executes Action A4 at step s. There is no change in any value of ξ.

Case 5. Suppose $x \in I_1$, and x executes Action A6 at step s. Then $x.lvl = 0$ and $Energy(x) \leq 2n$ by Lemma 4.5.

Case 6. Suppose $x \notin I_1$, and x executes Action A5 at step s. Let $y = x.par$, let $\ell = x.lvl$, and let $E = Energy(x)$ before the action, which implies that $Energy(y) \geq E + 2$ and $Energy(x) = E + 1$ after the action. Before the action, $\Xi \geq \xi(y) \geq \ell - 1 + E + 2$, while after the action $\xi(x) = \ell + E + 1$. Thus, the action does not cause Ξ to increase. □

Lemma 4.9 *There is no infinite acyclic execution of* DISJ.

Proof. By Lemma 4.8, there are only finitely many possible values of $\xi(x)$ starting from any given initial configuration, and thus there are only finitely many possible values of $x.lvl$. All other variables have only finitely many possible values. There are thus only finitely many possible states of any particular process in that execution, and hence only finitely many reachable configurations of DISJ starting from any given configuration. It follows that any infinite execution of DISJ must contain a cycle. □

Lemma 4.10 *There is no cyclic execution of* DISJ.

Proof. By contradiction. Suppose Γ is a computational cycle of DISJ. Then any change of variables at any step of Γ is reversed (meaning the effect of actions executed in the step is cancelled) by later steps of Γ.

Claim I: Γ contains no execution of Action A1, since that action is irreversible.

Claim II: Γ contains no execution of Action A3.

Proof of Claim II: Suppose the claim is false. Let ℓ be the minimum value of *lvl* assigned to any process by its execution of Action A3, and let x be a process such that $x.lvl \leftarrow \ell$ at some step of Γ. Since every step is reversible, x must execute Action A2 as well. Let $y = x.par$ at the step when x executes Action A3. By the minimality of ℓ, if $y.lvl < \ell$ at any step, then y never executes Action A3 or A2. It follows that $Level(x) = \ell$ at every step, which makes it impossible for x to execute Action A2, contradiction.

Claim III: Γ contains no execution of Action A2, by Claim II, since all steps must be reversible.

We now know that Γ contains no structure actions.

Claim IV: If x is not in a true tree, then x does not execute Action A5 or A6 during Γ.

Proof of Claim IV: If $x \in O_0$, we are done. Otherwise, the proof is by strong induction on $x.lvl$, which, by Claim II, never changes.

If x is not a true child, then x is invalid, and hence cannot execute Action A5. Otherwise, by the inductive hypothesis, $x.par$ never changes its color, which implies that x can change its color at most once. Since Γ is a cycle, that implies that x never changes color.

Claim V: If x is in a true tree T_r and one member of T_r never changes color, then x never changes color.

Proof of Claim V: If $x = y.par$, and if one of those two processes never changes color, the other can change color at most once. Since Γ is a cycle, it never changes color. By induction, no member of T_r can change color.

Claim VI: If T_r is a true tree and all members of T_r change color during Γ, then $x.done = 1$ for all $x \in T_r$.

Proof of Claim VI: If $x \in T_r$, then $Recruits(x) = \emptyset$ since x changes color twice. By bottom-up induction, $x.done = Done(x) = 1$ for all $x \in T_r$.

Claim VII: No process executes Action A5 or A6 during Γ.

Proof of Claim VII: By Claim IV, no process which is not in a true tree can change color. By Claim V, if any process in a true tree T_r changes color, then r changes color. By Claim VI, $r.done = 1$, contradiction.

Claim VIII: No process executes Action A4 during Γ.

Proof of Claim VIII: By bottom-up induction on *lvl*. No neighbor of x changes color (Claim VII), and by the inductive hypothesis, $y.done$ never changes for any $y \in True\_Chldrn(x)$. Therefore, $Done(x)$ never changes, hence $x.done$ can change at most once, which implies it never changes.

The statement of the lemma follows immediately from Claims I, II, III, IV, VII, and VIII. □

Lemma 4.11 *There is no infinite execution of* DISJ.

Proof. By Lemma 4.9, every infinite execution has a cycle, but by Lemma 4.10, there is no cyclic execution. □

Theorem 4.12 DISJ *is correct.*

Proof. By Lemma 4.11, every execution of DISJ reaches a terminal configuration. By Lemma 4.4, that configuration is legitimate. □

5 Round and Space Complexity

We first consider the case that $Output = 1$, that is, $I_1 \neq \emptyset$. For all x, let $L(x) = ||x, I_1||$, the value of $x.lvl$ in a legitimate configuration.

The stabilization time of DISJ is $O(n)$ rounds, as stated in Theorem 5.17. At first glance, that result appears to be obvious, yet we have not found a simple proof. Color waves, which we use to ensure convergence if $Output = 0$, greatly increase the difficulty of proving round complexity when $Output = 1$.

Initially, the value of $x.lvl$ could be anything. An execution of DISJ in the case that $Output = 1$ has three phases. In the first phase, processes where $x.lvl < L(x)$ are eliminated, and in the second phase, all values of $x.lvl$ are set to $L(x)$. In the third and last phase, $x.done \leftarrow 1$ for all x, and then all trees become color-locked.

In the case that $Output = 0$, there is only one phase, during which false trees are eliminated. The root of a false tree cannot absorb color waves, and thus, if the unfair daemon never selects a false root, the false roots will eventually be the only enabled processes, since false trees will become color-locked. The daemon is then forced to select false roots until legitimacy is achieved. This phenomenon is illustrated in Figure 7.

We define the set $\mathcal{S}$ recursively:

If $True\_Root\,(x)$ then $x \in \mathcal{S}$,

if $True\_Child\,(x) \wedge (p.par \in \mathcal{S}) \wedge (x.lvl = L(x))$ then $x \in \mathcal{S}$.

Lemma 5.1 *If $x \in O_1$ and t rounds have elapsed, then $x.lvl \geq \min\{L(x),\, t\}$.*

Proof. By induction on t. If $t = 0$, the lemma is trivial. Let $t > 0$.

Let step s be the last previous step at which x executed a structure action, or else the last step of the previous round, whichever is later.

Case 1. x has not executed a structure action since the end of round $t-1$. Then x was valid at the end of round $t-1$, since otherwise x would have executed Action A2 during the round.

Case 2. The most recent structure action of x was Action A3.

In either case, x had a parent at step s; let y be its parent at that step. At step s, $x.lvl = 1 + y.lvl \geq 1 + \min\{L(y),\, t-1\} \geq \min\{L(x),\, t\}$ by the inductive hypothesis and the triangle inequality; and $x.lvl$ has not changed since step s. □

Lemma 5.2 *If $x \in \mathcal{S}$ and $t \geq L(x)$ rounds have elapsed, then $x \in \mathcal{S}$ for the remainder of the execution.*

Proof. By induction on $L(x)$. If $L(x) = 0$, then $x \in I_1$, hence x is a true root and remains a member of $\mathcal{S}$.

Let $L(x) > 0$. Suppose that x executes a structure action at some step, say step s, after at least $L(x)$ rounds. Then there is some $y \in N(x)$ such that $y.lvl < L(x) - 1$ at step s. However, by Lemma 5.1, $y.lvl > \min\{L(y), t\} \geq L(x) - 1$ at step s, contradiction. □

We define a process x to be *closed* if $x \in \mathcal{S}$ and $\mathcal{S} \cap T_x = T_x^*$, where T_x^* is the subtree rooted at x in the terminal configuration (T_x is the set of descendents of x in T, see page 15 for the formal definition). Otherwise, we say x is *open*, *i.e.,* there are still processes that need to be recruited by T_x.

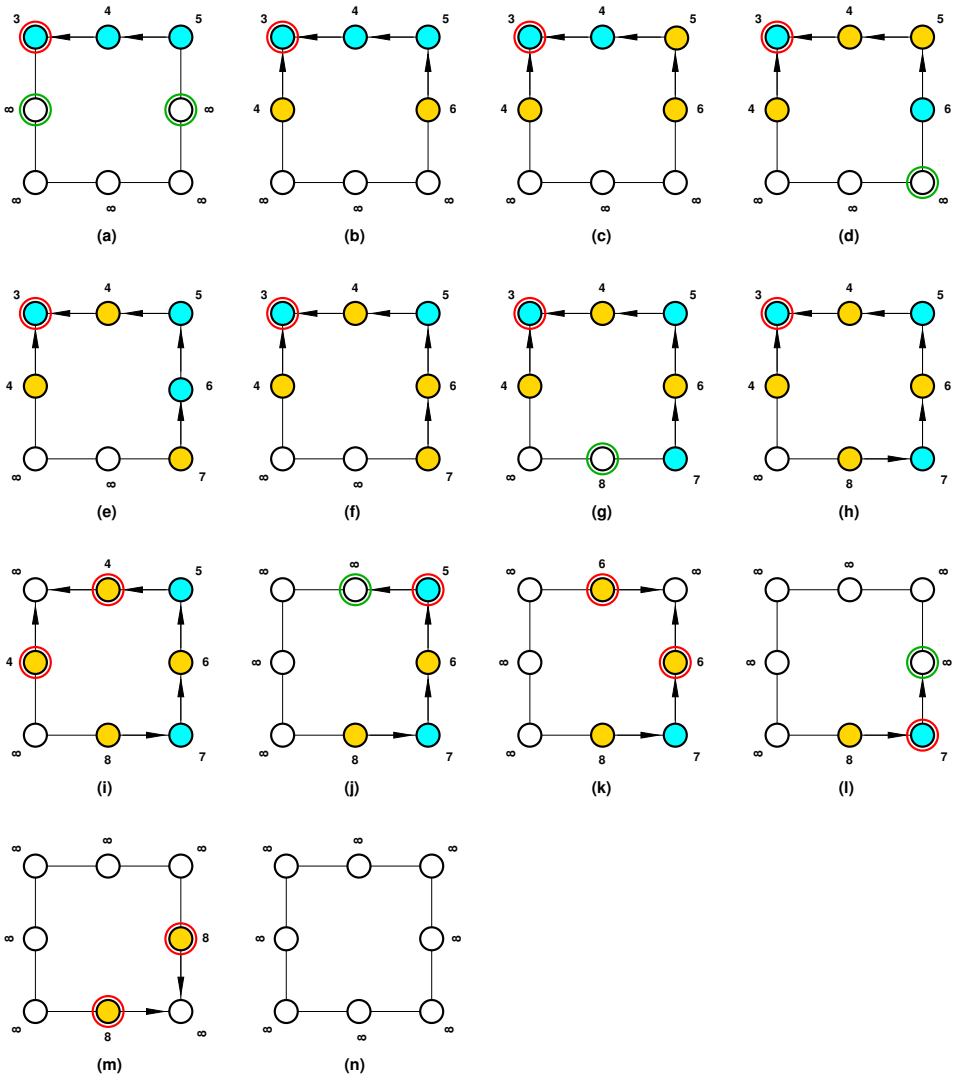


Figure 7: Elimination of a false tree when $Output = 0$. Numbers near processes indicate their levels. Color 0 for members of O_1 is represented by gold. Color 1 for members of O_1 is represented by cyan. Colors are irrelevant for members of O_0, so these processes are white. When an arrow is outgoing from a process, it designates its parent. When no arrow is outgoing from a process x, this means that $x.par = \perp$. A red circle indicates that a process is enabled to reset, while a green circle indicates a process is enabled to join. In this example, all enabled processes are selected at each step, except for the false root, the process at upper left, which is enabled to execute Action A2. The unfair daemon refuses to select that process, until the false tree is color-locked at (h). At that point, the false root is the only enabled process, and hence must be selected. The false tree is then eliminated in six steps.

The Benchmark Done-Closed We say that an execution satisfies the Done-Closed predicate if every member of I_1 is a true root, and if $x.done = 0$ for any $x \in \mathcal{S}$ which is open.

Lemma 5.3 *The Done-Closed predicate holds if at least $2\,\mathcal{D}$ rounds have elapsed.*

Proof. By Lemma 5.2, no process can leave $\mathcal{S}$.

Claim I: If $2\,\mathcal{D} - L(x)$ rounds have elapsed, then $x.done = 0$.

Proof of Claim I: By backwards induction on $L(x)$. If $L(x) = \mathcal{D}$, then x is closed, contradiction.

Suppose $L(x) > 0$. Pick $y \in \mathit{True\_Chldrn}^*(x)$ such that y is open. By the inductive hypothesis, either $y \notin \mathcal{S}$ or $y.done = 0$, and hence $\mathit{Done}\,(x) = 0$, if $2\,\mathcal{D} - L(y)$ rounds have elapsed. Within one more round, $x.done = 0$.

The lemma follows immediately from Claim I, since $L(x) \leq \mathcal{D}$ for all x. □

To complete the proof of round complexity, we need to define

$$\lambda(x) = \begin{cases} \lambda(x.par) \text{ if } \mathit{True\_Child}\,(x) \wedge (x.color \neq x.par.color) \\ 1 + \lambda(x.par) \text{ if } \mathit{True\_Child}\,(x) \wedge (x.color = x.par.color) \\ 0 \text{ otherwise} \end{cases}$$

That is, $\lambda(x)$ is the number of color alternations in the directed chain from x to r, where r the root of the tree to which x belongs.

Remark 5.4 *If x is a true child, then $\lambda(x.par) \leq \lambda(x) \leq 1 + \lambda(x.par)$.*

Remark 5.5 *If $x \in O_1$, then $\lambda(x) \leq x.lvl$.*

5.1 Achieving Correct Levels

The main problem in proving that levels converge to their correct values is the possibility that, in the initial configuration, a tree T_r could contain a chain whose length is greater than $\mathcal{D}$, where $\Omega(n)$ processes in the chain have color 0, and none are enabled to reset. The same rules that prevent a false tree from growing forever cause r to wait $\Omega(n)$ rounds, in the worst case, before executing Action A6 for the first time. An illustrative example is given in Figure 8. In this example, the white node x will eventually join the doubly circled node y by Action A3. However, this will happen only after y changes its color to 1 by absorbing a color wave of color 1 with Action A6. Now, this color wave must be initiated by Action A5 at the process of height 5 and then propagated up following parent pointers, by Action A5, until reaching y.

If Done-Closed holds, let $\mathit{Num\_Clr\_Chng}\,(x)$ be the number of times x has executed Action A3, A5, or A6 since Done-Closed was first achieved.

If Done-Closed, let $\pi(x) = 4\,\mathcal{D} - 2\,\mathit{Num\_Clr\_Chng}\,(r) + x.lvl - 2\,\lambda(x)$ for any $r \in I_1$ and any $x \in T_r$. Then let $\Pi(r) = \max\{\pi(x) : x \in T_r\}$ for any $r \in I_1$. Note that $\Pi(r) = O(n)$.

Lemma 5.6 *If Done-Closed holds and $r \in I_1$, then $\Pi(r)$ does not increase.*

Proof. Suppose $\Pi(r)$ increases at step s. We can assume that exactly one process, x executes at that step. If x is not a member of T_r either before or after step s, then the action has no effect on $\Pi(r)$. Since Done-Closed holds, x cannot execute Action A1. If x executes Action A4, there is no effect on $\Pi(r)$. We now consider the remaining cases. Let $N = \mathit{Num\_Clr\_Chng}\,(r)$ before step s.

Case 1. x executes Action A3 at step s, choosing y to be its parent, where $y \in T_r$. Let $\ell = y.lvl$, and let $\Lambda = \lambda(y)$. Then $\Pi(r) \geq \pi(y) = 2\,\mathcal{D} - 2\,N + \ell - 2\,\Lambda$ before the step. After the step $\pi(x) = 2\,\mathcal{D} - 2\,N + \ell - 2\,\Lambda - 1$, and thus the action has no effect on $\Pi(r)$.

Case 2. $x \in T_r$ before the step, and leaves T_r at the step, either by executing Action A2 or by executing Action A3, choosing a parent not in T_r. The processes which were in T_x are no longer in T_r after the action, but the values of $\pi(y)$ are unaffected for $y \in T_r \setminus T_x$. Thus $\Pi(r)$ cannot increase.

Case 3. x executes Action A5, and $x \neq r$. Then $\pi(x)$ decreases by 2 during the step since $\lambda(x)$ increases by 1, while $\pi(y)$ is unchanged for all other $y \in T_r$. Thus, $\Pi(r)$ does not increase.

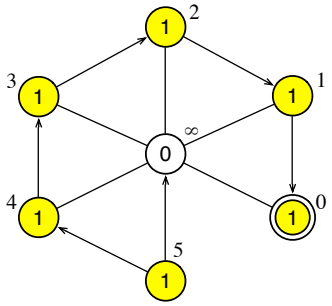


Figure 8: Example showing that color waves slow the growth of true trees. The doubly circled process is the sole member of I_1. The internal bit is the value of $x.out$. Numbers near processes indicate their levels. Color 0 for members of O_1 is represented by gold. Color 1 for members of O_1 is represented by cyan. Colors are irrelevant for members of O_0. When an arrow is outgoing from a process, it designates its parent. When no arrow is outgoing from a process x, this means that $x.par = \perp$.

Case 4. r executes Action A6 at step s. Before the step, $\pi(r) = 2\mathcal{D} - 2N$, while after the step, $\pi(r) = 2\mathcal{D} - 2N - 2$, since $Num\_Clr\_Chng(r)$ increments. The value of $\pi(y)$ is unchanged for all other $y \in T_r$, since N increments and $\lambda(y)$ decrements. Thus $\Pi(r)$ does not increase. □

If Done-Closed holds and $r \in I_1$, we define a predicate $\mathcal{X}(r)$ as follows.

$$\mathcal{X}(r) = \begin{cases} 1 \text{ if } \exists x \in T_r : \pi(x) = \Pi(r) \wedge x.color = 1 \wedge Recruits(x) \neq \emptyset \\ 0 \text{ otherwise} \end{cases}$$

Lemma 5.7 *If Done-Closed holds and $r \in I_1$, then $\Pi(r)$ decreases during the next two rounds.*

Proof. Claim I: If $\mathcal{X}(r) = 0$, then $\Pi(r)$ decreases during the next round.

Proof of Claim I: Let $X = \{x \in T_r : \pi(x) = \Pi(r)\}$. For all $x \in X$, either $x = r$ or $x.par.color = x.color$, since otherwise $\pi(x.par) > \pi(x)$, and $y.color \neq x.color$ for all $y \in True\_Chldrn(x)$, since otherwise $\pi(y) > \pi(x)$; furthermore, either $x.color = 0$ or $Recruits(x) = \emptyset$, since $\mathcal{X}(r) = 0$. Each member of X is thus enabled to execute Action A5, and will do so within one round. The effect of these actions is to cause $\Pi(r)$ decrease. By Lemma 5.6, no other actions during that round can cause $\Pi(r)$ to increase. This proves Claim I.

Claim II: If $\mathcal{X}(r) = 1$, then within one round either $\Pi(r)$ decreases or $\mathcal{X}(r)$ becomes 0.

Proof of Claim II: Suppose $\Pi(r)$ does not decrease during the next round. Let X be the set of all $x \in T_r$ such that $\pi(x) = \Pi(r)$ and x does not execute Action A5 during the next round. If $x \in X$ and $y \in Recruits(x)$, y will execute a structure action during the next round, and will no longer be a member of $Recruits(x)$. Thus, $\mathcal{X}(r) = 0$ after one round. This proves Claim II.

The lemma follows immediately from Claims I and II, and Lemma 5.6. □

Lemma 5.8 *If Done-Closed holds, $r \in I_1$, and $Num\_Clr\_Chng > 2\mathcal{D}$, then $T_r^* \subseteq T_r$.*

Proof. Let $x \in T_r^* \setminus T_r$. Let $x = x_k, x_{k-1}, \ldots x_1, x_0 = r$ be the chain of ancestors of x in T_r^*, and let ℓ be maximum such that $x_\ell \in T_r$ when the predicate Done-Closed is first achieved, which we say is at step s_ℓ. For each $\ell < i \leq k$, x_i will join T_r at a subsequent step, say step s_i. For $\ell \leq i \leq k$, let $\lambda_i = \lambda(x_i)$ at step s_i, and let N_i be the value of $Num\_Clr\_Chng(r)$ at step s_i. Note that $N_\ell = 0$.

Claim I: $\lambda_i - \lambda_{i-1} + (N_i + N_{i-1}) = 2$ if $\ell < i \leq k$, and

$$\lambda_{\ell+1} - \lambda_\ell + N_{\ell+1} = \begin{cases} 2 \text{ if } x_\ell.color = 0 \text{ at } s_\ell \\ 1 \text{ if } x_\ell.color = 1 \text{ at } s_\ell \end{cases}$$

Proof of Claim I: Refer to the chain of processes, to r from the lowest ancestor of x currently in T_r, as the *current chain*. Then λ_i is the number of color alternations in the current chain at step s_i. Each

time r executes Action A5, it absorbs one color wave, hence decrements the number of color alternations in the current chain. At step s_i, that number of color changes is incremented as x_i joins the string, but there is another incrementation when x_{i-1} changes color to 1 before that step by executing Action A5, since $x_{i-1}.color = 0$ after step s_{i-1}. The net incrementation of the number of color alternations in the chain is thus $2 - (N_i - N_{i-1})$, except for the special case that $i = \ell + 1$ and $x_\ell.color = 1$ at step s_ℓ, in which case it is $1 - N_{\ell+1}$. This proves Claim I.

Claim II: $N_k \leq 2k$.

Proof of Claim II: From Claim I, and since $\lambda_\ell \leq \ell$, we have

$$N_k \leq 2k - 2\ell - \lambda_k + \lambda_\ell \leq 2k - 2\ell + \lambda_\ell \leq 2k$$

By Claim II, $N_k \leq 2\,\mathcal{D}$. Thus, $x \in T_r$ if r has executed Action A5 more than $2\,\mathcal{D}$ times. □

Lemma 5.9 *If Done-Closed holds, $r \in I_1$, and $\Pi(r) < 0$, then $T_r^* \subseteq T_r$.*

Proof. Since $r.lvl = \lambda(r) = 0$, we have $4\,\mathcal{D} - 2\,\mathit{Num\_Clr\_Chng}\,(r) = \pi(r) \leq \Pi(r) < 0$, hence $\mathit{Num\_Clr\_Chng}\,(r) > 2\,\mathcal{D}$. We are done by Lemma 5.8. □

We say that *levels are correct* if all processes are valid and $x.lvl = L(x)$ for all x.

From Lemma 5.9, we immediately have:

Corollary 5.10 *If Done-Closed holds and $\Pi(r) \leq 0$ for all $r \in I_1$, then levels are correct.*

Lemma 5.11 *If Output $= 1$, then levels are correct within $O(n)$ rounds.*

Proof. Since $\Pi(r) = O(n)$ for all r, the result follows from Lemmas 5.7 and 5.9. □

5.2 Achieving Color-Lock

If *Output* $= 1$, we say that *processes are done* if levels are correct and $x.done = 1$ for all x. In this subsection, we prove that, once levels are correct, all processes are done within $O(\mathcal{D})$ rounds in a convergecast wave in each tree, after which all trees are color-locked within $O(\mathcal{D})$ additional rounds, since color waves are continually initiated at the leaves but never absorbed by the roots, and cannot pass each other.

Lemma 5.12 *If Output $= 1$ and levels are correct, then processes are done within $\mathcal{D} + 1$ additional rounds.*

Proof. Suppose *Output* $= 1$ and levels are correct.

Claim: Within $t + 1$ additional rounds, $x.done$ if $L(x) \geq \mathcal{D} - t$.

Proof of Claim I: By induction on t. If $t = 0$, then $\mathit{True\_Chldrn}\,(x) = \emptyset$, hence $\mathit{Done}\,(x) = 1$ after t rounds, and thus $x.done$ within $t + 1$ rounds.

Let $t > 0$. After t rounds, by the inductive hypothesis, $y.done$ for all $y \in \mathit{True\_Chldrn}\,(x)$, and thus $\mathit{Done}\,(x) = 1$. Thus, $x.done$ within $t + 1$ rounds. This proves the claim.

The Lemma follows from the claim, by setting $t = \mathcal{D}$. □

Lemma 5.13 *If processes are done and x is any process, then at any step*

(a) *$\lambda(x)$ increases if x executes Action* A5*,*

(b) *$\lambda(x)$ does not change otherwise.*

Proof. Suppose x executes at step s. Since processes are done, x is a true child, and the only action x can execute is Action A5.

Let $y = x.par$. By definition $\lambda(y)$ does not change at the step. Let $\Lambda = \lambda(x)$ before the step. By the guard of Action A5, $\lambda(y) = \Lambda$ and $\lambda(z) = \Lambda + 1$ for all $z \in \mathit{True\_Chldrn}\,(x)$. Neither y nor any child of x is enabled to execute during the step, and by the statement of Action A5, $\lambda(x) = \Lambda + 1$ after the step. Since no child of x is affected by the action of x, no other descendant of x is affected, hence no process other than x is affected. □

Lemma 5.14 *If t rounds have elapsed since processes were done*

$$\lambda(x) \geq \min\left\{x.lvl,\ \left\lceil (x.lvl + t - \mathcal{D})/2 \right\rceil\right\}$$

Proof. By double induction on t and $x.lvl$. Write $N = x.lvl + t - \mathcal{D}$. Recall that $x.lvl \leq \mathcal{D}$.

If $x.lvl = 0$ we are done. If $t = 0$, we are done since $x.lvl \leq \mathcal{D}$.

Let $t > 0$ and $x.lvl > 0$.

Case 1. N is odd. After $t-1$ rounds, by the inductive hypothesis,

$$\lambda(x) \geq \min\left\{x.lvl,\ \left\lceil (N-1)/2 \right\rceil\right\} = \left\{x.lvl,\ \left\lceil N/2 \right\rceil\right\}$$

We are done by Lemma 5.13.

Case 2. $\lceil (N-1)/2 \rceil \geq x.lvl$. By Remark 5.5 and the inductive hypothesis, $\lambda(x) = x.lvl$ after $t-1$ rounds have elapsed. By Lemma 5.13 and Remark 5.5, we are done.

Case 3. N is even and $\lceil (N-1)/2 \rceil < x.lvl$, hence $N/2 \leq x.lvl$. We need to show that $\lambda(x) \geq N/2$. If $\lambda(x) \geq N/2$ after $t-1$ rounds, we are done by Lemma 5.13. Otherwise, after $t-1$ rounds, by Remark 5.4 and the inductive hypothesis, $\lambda(x.par) = \lambda(x) = N/2 - 1$, while $\lambda(y) = N/2$ for all $y \in \mathit{True\_Chldrn}(x)$; thus x is enabled to execute Action A5. Within one more round, x will execute that action, and $\lambda(x)$ will increase to $N/2$. □

Lemma 5.15 *If Output $= 1$, then* DISJ *reaches a legitimate configuration within $O(n)$ rounds.*

Proof. By Lemma 5.11, levels are correct within $O(n)$ rounds. By Lemma 5.12, processes are done within $O(\mathcal{D})$ additional rounds. By Lemma 5.14 and Remark 5.5, $\lambda(x) = x.lvl$ for all x within $O(\mathcal{D})$ additional rounds. At that point, no process is enabled, and hence the configuration is legitimate by Lemma 4.4. The round complexity of the execution is thus $O(n)$. □

We next consider the case that *Output* $= 0$, *i.e.,* that $I_1 = \emptyset$. The configuration is legitimate if $O_1 = \emptyset$, but in the initial configuration, O_1 need not be empty. Our goal is to show that O_1 becomes empty within $O(n)$ rounds. We make use of the function *Energy*, as defined in Section 4.2, as well as Lemma 4.5 from that section.

The next lemma is, essentially, Lemma 7.7 of [16].

Lemma 5.16 *If Output $= 0$ and Max_Energy > 0, then Max_Energy decreases during the next round.*

Proof. Since $I_1 = \emptyset$ there are no true roots, but if $O_1 \neq \emptyset$ there must be at least one false root. Furthermore, if $\mathit{Energy}(x) = \mathit{Max\_Energy}$, then x is a false root.

Claim I: *Max_Energy* does not increase.

Proof of Claim I: Consider an execution of a process x. We can assume that no other process executes at the same step. Since $I_1 = \emptyset$, x cannot execute Action A1. If x executes Action A4, there is no effect on *Max_Energy*.

Suppose x executes Action A3, choosing y to be its parent. Before that action, $y.color = 1$, hence $\mathit{Energy}(y) \geq 2$, while after the action, $\mathit{Energy}(x) = 1$. Thus, that action does not cause *Max_Energy* to increase.

Suppose x executes Action A2. Then $\mathit{Energy}(y)$ decreases if y is an ancestor of x; otherwise $\mathit{Energy}(y)$ is unaffected for any other process y.

Suppose x executes Action A5. Then x is a true child, since there is true root. Let $y = x.par$, and let $E = \mathit{Energy}(x)$ before the step. Then $\mathit{Energy}(y) \geq E+2$ and $\mathit{Energy}(z) \leq E-1$ for all $z \in \mathit{True\_Chldrn}(x)$. After the step, $\mathit{Energy}(x) = E+1$, while the energy of no other process changes.

Thus, in all cases an action does not cause *Max_Energy* to increase. This completes the proof of Claim I.

If $\mathit{Energy}(x) = \mathit{Max\_Energy}$, then x is a false root, hence enabled to execute Action A2. Within one round, all such processes will execute, causing *Max_Energy* to decrease. □

Theorem 5.17 *The stabilization time of* DISJ *is* $O(n)$ *rounds.*

Proof. If $Output = 1$, then DISJ converges within $O(n)$ rounds by Lemma 5.15. Otherwise, $Max\_Energy = 0$ within $O(n)$ rounds, by Lemmas 4.5 and 5.16, hence $O_1 = \emptyset$ by the definition of *Energy*. The configuration is then legitimate. □

5.3 Space Complexity

For any process x, let $\delta(x)$ be the degree of x, *i.e.,* the number of neighbors of x. we define $C(x)$ as follows. If $Output = 0$, let $C(x) = \delta(x) + 5$ for all x. If $Output = 1$, let $C(x) = \lceil \log_2 (L(x) + 2) \rceil + \delta(x) + 4$.

Theorem 5.18 DISJ *has space complexity* $C(x)$.

Proof. Each process x needs 4 bits to hold the values of $x.in$, $x.out$, $x.color$, and $x.done$, plus one bit dedicated to each of its neighbors, to store the value of $x.par$.

If $Output = 1$, x must store values of $x.lvl$ up to $L(x)$. Since ∞ is an option the number of possible values of $x.lvl$ needed is $L(x) + 2$. On the other hand, if $Output = 0$, we can store the value of $x.lvl$ with only one bit.

We need to show that, if the memory of each x consists of at least $C(x)$ bits, there is no configuration of DISJ which is deadlocked for lack of space. We will show that, if any processes are blocked, there is another process which is not blocked and is enabled to execute Action A2.

Suppose $Output = 1$. The only action that could cause a process to overflow its memory is Action A3. Suppose x is enabled to execute an action which overflows its memory. Then x must have a neighbor y such that $y.lvl > L(y)$. Pick z such that $z.lvl$ is minimum subject to the condition that $z.lvl > L(z)$. Then z is invalid, which implies that z is enabled to execute Action A2, and cannot be blocked because A2 has priority over A3.

On the other hand, suppose there is a configuration at which some process is blocked and $Output = 0$. Let x be the process with the smallest value of $x.lvl$. Then x is enabled to execute Action A2. □

Since $C(x) = O(\log \mathcal{D} + \Delta)$, we can also say that DISJ has space complexity $O(\log \mathcal{D} + \Delta)$.

6 Conclusion

We introduced a new problem, called the *disjunction* problem. A solution to this problem can be used to detect disconnected components, or to cluster a network, for example.

We give a silent self-stabilizing algorithm for the disjunction problem, assuming a connected anonymous bidirectional network. Our algorithm is designed in the composite atomicity model, assuming the distributed unfair daemon (the most general daemon).

Our algorithm stabilizes in $O(n)$ rounds, where n is the size of the network. Moreover, it requires $O(\log \mathcal{D} + \Delta)$ bits per process, where $\mathcal{D}$ and Δ are, respectively, the diameter, and the maximum degree of the network.

References

[1] A. K. Datta, S. Devismes, L. L. Larmore, Self-stabilizing silent disjunction in an anonymous network, in: D. Frey, M. Raynal, S. Sarkar, R. K. Shyamasundar, P. Sinha (Eds.), ICDCN, 14th International Conference on Distributed Computing and Networking, Vol. 7730 of Lecture Notes in Computer Science, Springer, Mumbai, India, 2013, pp. 148–160.

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

[3] S. Dolev, Self-stabilization, MIT Press, 2000.

[4] S. Dolev, M. G. Gouda, M. Schneider, Memory Requirements for Silent Stabilization, Acta Inf. 36 (6) (1999) 447–462.

[5] N.-S. Chen, H.-P. Yu, S.-T. Huang, A self-stabilizing algorithm for constructing spanning trees, Information Processing Letters 39 (1991) 147–151.

[6] Z. Collin, S. Dolev, Self-stabilizing depth-first search, Information Processing Letters 49 (1994) 297–301.

[7] J. Burman, S. Kutten, Time Optimal Asynchronous Self-stabilizing Spanning Tree, in: Distributed Computing, 21st International Symposium (DISC), 2007, pp. 92–107.

[8] A. Kravchik, S. Kutten, Time Optimal Synchronous Self Stabilizing Spanning Tree, in: DISC, 2013, pp. 91–105.

[9] A. K. Datta, L. L. Larmore, P. Vemula, A Self-Stabilizing $O(k)$-Time k-Clustering Algorithm, The Computer Journal (2009) bxn071.

[10] A. K. Datta, S. Devismes, K. Heurtefeux, L. L. Larmore, Y. Rivierre, Competitive self-stabilizing k-clustering, Theor. Comput. Sci. 626 (2016) 110–133.

[11] C. Johnen, F. Mekhaldi, Self-stabilizing with service guarantee construction of 1-hop weight-based bounded size clusters, J. Parallel Distrib. Comput. 74 (1) (2014) 1900–1913.

[12] C. Glacet, N. Hanusse, D. Ilcinkas, C. Johnen, Disconnected components detection and rooted shortest-path tree maintenance in networks, in: SSS'2014, 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems, LNCS, Paderborn, Germany, 2014, pp. 120–134.

[13] A. Leon-Garcia, I. Widjaja, Communication Networks, 2nd Edition, McGraw-Hill, Inc., New York, NY, USA, 2004.

[14] D. Angluin, Local and global properties in networks of processors (extended abstract), in: The 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 82–93.

[15] S. Dolev, A. Israeli, S. Moran, Self-stabilization of dynamic systems assuming only Read/Write atomicity, Dist. Comp. 7 (1) (1993) 3–16.

[16] A. K. Datta, L. L. Larmore, P. Vemula, Self-stabilizing leader election in optimal space under an arbitrary scheduler, Theor. Comput. Sci. 412 (40) (2011) 5541–5561.

[17] Y. Afek, A. Bremler-Barr, Self-stabilizing unidirectional network algorithms by power supply, Chicago J. Theor. Comput. Sci. 1998.

[18] S.-T. Huang, N.-S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, Inf. Process. Lett. 41 (2) (1992) 109–117.

[19] A. Cournier, S. Devismes, V. Villain, Light enabling snap-stabilization of fundamental protocols, ACM Transactions on Autonomous and Adaptive Systems 4 (1).

[20] A. Cournier, S. Rovedakis, V. Villain, The first fully polynomial stabilizing algorithm for BFS tree construction, in: the 15th International Conference on Principles of Distributed Systems (OPODIS'11), Springer LNCS 7109, 2011, pp. 159–174.

[21] S. Devismes, C. Johnen, Silent self-stabilizing bfs tree algorithms revisited, Journal of Parallel and Distributed Computing 97 (2016) 11 – 23.

[22] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, Time Optimal Self-stabilizing Synchronization, in: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC), 1993, pp. 652–661.

[23] S. Dolev, T. Herman, Superstabilizing Protocols for Dynamic Distributed Systems, Chicago J. Theor. Comput. Sci. 1997.

[24] A. Arora, M. G. Gouda, Distributed Reset, IEEE Trans. Computers 43 (9) (1994) 1026–1038.

[25] A. K. Datta, L. L. Larmore, H. Piniganti, Self-stabilizing Leader Election in Dynamic Networks, in: Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium (SSS), 2010, pp. 35–49.

[26] A. K. Datta, L. L. Larmore, P. Vemula, Self-stabilizing Leader Election in Optimal Space under an Arbitrary Scheduler, Theor. Comput. Sci. 412 (40) (2011) 5541–5561.

[27] A. K. Datta, L. L. Larmore, P. Vemula, An O(n)-time Self-stabilizing Leader Election Algorithm, J. Parallel Distrib. Comput. 71 (11) (2011) 1532–1544.

[28] K. Altisen, A. Cournier, S. Devismes, A. Durand, F. Petit, Self-stabilizing leader election in polynomial steps, Information and ComputationTo appear.

[29] A. K. Datta, L. L. Larmore, S. Devismes, K. Heurtefeux, Y. Rivierre, Self-stabilizing small k-dominating sets, International Journal of Networking and Computing 3 (1) (2013) 116–136.

[30] C. Johnen, Fast, silent self-stabilizing distance-k independent dominating set construction, Inf. Process. Lett. 114 (10) (2014) 551–555.

[31] C. Johnen, Memory efficient self-stabilizing distance-k independent dominating set construction, in: A. Bouajjani, H. Fauconnier (Eds.), Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers, Vol. 9466 of Lecture Notes in Computer Science, Springer, 2015, pp. 354–366.

[32] E. Caron, A. K. Datta, B. Depardon, L. L. Larmore, A self-stabilizing k-clustering algorithm for weighted graphs, J. Parallel Distrib. Comput. 70 (11) (2010) 1159–1173.

[33] A. K. Datta, S. Devismes, L. L. Larmore, A Self-Stabilizing $O(n)$-Round k-Clustering Algorithm, in: International Symposium on Reliable Distributed Systems (SRDS), 2009, pp. 147–155.

[34] S. Dolev, A. Israeli, S. Moran, Uniform dynamic self-stabilizing leader election, IEEE Trans. Parallel Distrib. Syst. 8 (4) (1997) 424–440.

[35] B. Awerbuch, B. Patt-shamir, G. Varghese, Self-stabilization by local checking and correction (extended abstract), in: In Proceedings of 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1991, pp. 268–277.