Self-Stabilizing Silent Disjunction in an Anonymous Network

Ajoy K. Datta[1], Stéphane Devismes[2], and Lawrence L. Larmore[1]

[1] Department of Computer Science, University of Nevada Las Vegas, USA
[2] VERIMAG UMR 5104, Université Joseph Fourier, France

Abstract. Given a fixed *input bit* to each process of a connected network of processes, the *disjunction problem* is for each process to compute an *output bit*, whose value is 0 if all input bits in the network are 0, and 1 if there is at least one input bit in the network which is 1. A uniform asynchronous distributed algorithm DISJ is given for the disjunction problem in an anonymous network. DISJ is self-stabilizing, meaning that the correct output is computed from an arbitrary initial configuration, and is silent, meaning that every computation of DISJ is finite. The time complexity of DISJ is $O(n)$ rounds, where n is the size of the network. DISJ works under the *unfair daemon*.

Keywords: anonymous, disjunction, self-stabilization, silence, unfair daemon.

1 Introduction

Given a network of processes $\mathcal{G}$, where each process has a fixed *input bit*, $\mathit{Input}(x)$, the *disjunction problem* is for each process to compute $\mathit{Output} = \bigvee_{x \in \mathcal{G}} \mathit{Input}(x)$, the disjunction of all input bits in the network.

A *distributed solution* to the disjunction problem is a distributed algorithm which computes an *output bit* for each process, such that all output bits are equal to *Output*. The solution given in this paper, the distributed algorithm DISJ, correctly solves the disjunction problem if the network is connected. DISJ is self-stabilizing [1, 2], meaning that a correct output configuration is reached in finite time after arbitrary initialization, and is silent, meaning that eventually the computation of DISJ will halt. DISJ works under the *unfair* scheduler (daemon).

DISJ is uniform, meaning that every process has the same program, and is anonymous, meaning that processes are not required to have distinguished IDs. The *round complexity* of DISJ is $O(n)$, where n is the size of the network. We use the composite model of computation [2].

1.1 Related Work

We are not aware of closely related work in the literature. Although we use some of the same techniques in this paper that are used for leader election, the disjunction problem in an anonymous network cannot be solved by using a leader election algorithm, nor by using an algorithm to construct a spanning tree. In fact, there is no distributed algorithm which elects a leader or which constructs a spanning tree for general anonymous networks.

1.2 Outline of the Paper

In Section 2, we explain our model of computation. In Section 3, we give the formal definition of DISJ. In Section 4, we sketch the proof of the self-stabilization, silence, and time complexity of DISJ.

2 Preliminaries

We assume that we are given an anonymous network of processes. Let $N(x)$ be the set of *neighbors* of a process x.

A *self-stabilizing* [1, 2] 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 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 x is a Boolean expression involving the registers of x and its neighbors. The *statement* of an action of x updates one or more variables of x. 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.

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 $\gamma_p \mapsto \gamma_{p+1} \ldots \mapsto \gamma_q$ such that each $\gamma_i \mapsto \gamma_{i+1}$ is a step.

We measure the time complexity of DISJ in *rounds* [2]. We say that a finite computation $\varrho = \gamma_p \mapsto \gamma_{p+1} \mapsto \ldots \mapsto \gamma_q$ is a *round* if every process which is enabled at γ_p is either neutralized or executes an action at some step, and if the computation $\gamma_p \mapsto \gamma_{p+1} \mapsto \ldots \mapsto \gamma_{q-1}$ does not satisfy that condition. We define the *round complexity* of a computation to be the number of disjoint rounds in the computation.

3 DISJ

In this section, we give the formal definition of our algorithm, DISJ, which solves the disjunction problem in an anonymous connected network, $\mathcal{G}$. The fundamental idea of DISJ is to build a *local BFS tree* rooted at every process whose input bit is 1. Each process will join the tree rooted at the nearest process with input bit 1; ties will be broken arbitrarily. The construction of the BFS trees is by flooding.

The main difficulty with this method is the possibility that, in the initial configuration (which is arbitrary) there could be "fictitious" BFS trees. It is necessary to delete all such fictitious trees. This is an easy task if $\mathit{Output} = 1$, but is difficult if $\mathit{Output} = 0$. If a process does not have a parent in one of the trees, it will delete itself from the structure. Our problem is to ensure that a fictitious tree does not grow as fast at the leaf end as it deletes itself from the root end.

The method we use to ensure deletion of fictitious trees is derived from the *color wave* method of [3]. Each process in a tree or fictitious tree, whether true or fictitious, has a *color*, either 0 or 1. A process can only recruit a new process to the tree if its color is 1, and the recruited process will initially have color 0. Colors change in pipelined convergecast waves. Colors in a tree must alternate, and to allow the color waves to continue upward, the root of each BFS tree (a process whose input bit is 1) must "absorb" each wave. A fictitious tree will not be rooted at a process with input bit 1, and thus color waves will not be absorbed. "Color lock," the situation where the waves are maximally crowded and cannot move up, will eventually stop the growth of the fictitious tree.

We use the concept of *energy* introduced in [3]. $\mathit{Energy}(x)$ is a positive integer for each process x whose output bit is 1, and zero for processes with output bit 0. If $\mathit{Output} = 0$, $\mathit{Energy}(x) \leq 2n$ for all x, and the maximum value of *Energy* decreases by at least 1 during every round, and thus must eventually reach zero. At that point, every process has output bit 0. In one more round, DISJ converges.

3.1 Definition of DISJ

Recall that $\mathit{Input}(x)$ is the input bit of the process x, and that *Output* is the disjunction of all input bits. Define $I_i = \{x : \mathit{Input}(x) = i\}$, for $i = 0, 1$. Thus, $\mathit{Output} = 0$ if $I_1 = \emptyset$, while $\mathit{Output} = 1$ 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 $L(x) = ||I_1, x||$.

Variables of DISJ*.* Each process x has the following variables.

1. $x.\mathit{out} \in \{0, 1\}$, the *output bit* of x.
   When DISJ halts, $x.\mathit{out} = \mathit{Output}$ for all x. We let $O_i = \{x : x.\mathit{out} = i\}$, for $i = 0, 1$. During a computation of DISJ, the sets O_i can change.
2. $x.\mathit{parent} \in N(x) \cup \{\perp\}$, the *parent* of x.
   If $\mathit{Output} = 0$, then $x.\mathit{parent} = \perp$ for all x when DISJ halts. If $\mathit{Output} = 1$, then, when DISJ halts, $x.\mathit{parent} = \perp$ for all $x \in I_1$, while $x.\mathit{parent}$ is the parent pointer of x in the local BFS tree rooted at the nearest member of I_1 if $x \in I_0$.
3. $x.\mathit{level} \geq 0$, integer or ∞, the *level* of x.
4. $x.\mathit{color} \in \{0, 1\}$, the *color* of x.
   If $x.\mathit{out} = 0$, the value of $x.\mathit{color}$ is irrelevant. The purpose of the color variable is to ensure that eventually $x.\mathit{out} = 0$ for all x, if $\mathit{Output} = 0$. The main difficulty of the problem in that case is eliminating processes with output bit 1. We accomplish this task by using *color waves*, which ensure that "fictitious" trees shrink faster than they grow.
5. $x.\mathit{done}$, Boolean. This variable is irrelevant if $x.\mathit{out} = 0$. If $\mathit{Output} = 1$, the variable *done* is used to achieve silence when all BFS trees have been constructed. In that case, $x.\mathit{done} = \textsc{true}$ for all x when DISJ halts.

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

1. $\mathit{Level}(x) = \begin{cases} 0 \textbf{ if } \mathit{Input}(x) = 1 \\ \infty \textbf{ if } \mathit{Input}(x) = 0 \text{ and } N(x) \subseteq O_0 \\ 1 + \min\{y.\mathit{level} : y \in N(x) \cap O_1\} \textit{ otherwise} \end{cases}$
   When DISJ halts, $x.\mathit{level} = \mathit{Level}(x) = L(x)$ for all x.
2. $\mathit{Chldrn}(x) = \begin{cases} \{y \in N(x) \cap O_1 : y.\mathit{parent} = x \text{ and } y.\mathit{level} = 1 + x.\mathit{level}\} \textbf{ if } x \in O_1 \\ \emptyset \textbf{ if } x \in O_0 \end{cases}$
   the *children* of x in its local BFS tree.
3. $0\_\mathit{Valid}(x)$, Boolean, meaning that x is in a valid state with output bit 0, which is true if and only if all the following conditions hold.
   (a) $x \in O_0$

   (b) $x.level = Level(x)$
   (c) $x.parent = \bot$
4. $1\_Valid(x)$, Boolean, meaning that x is in a valid state with output bit 1, which is true if and only if all the following conditions hold.
   (a) $x \in O_1$
   (b) $x.level = Level(x)$
   (c) If $Input(x) = 0$ then $x.parent \in O_1$ and $x.level = 1 + x.parent.level$.
   (d) If $Input(x) = 1$ then $x.parent = \bot$ and $x.level = 0$.
   (e) If $y \in N(x)$ then $y.level + 1 \geq x.level$.
5. $Valid(x) \equiv 0\_Valid(x) \lor 1\_Valid(x)$, Boolean, meaning that x is *valid.* If $Valid(x) = $ FALSE, we say x is *invalid.*
   An invalid process x is enabled to execute the Reset action, A1, which causes x to become valid.
6. $Can\_Recruit(x)$, Boolean, meaning that there is a neighbor of x which can be recruited by x. This function is TRUE if and only $1\_Valid(x)$ and there is some $y \in N(x) \cap O_0$ such that $y.level = x.level + 1$.
7. $Done(x)$, Boolean, indicates that there should be no further recruitment of processes by x or any descendant of x in its local BFS tree.
   This function is TRUE if and only if $1\_Valid(x)$, $Can_Recruit(x) = $ FALSE, and $y.done$ for all $y \in Chldrn(x)$.

Actions of DISJ

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*. The second column gives an informal name of the action.

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, there is an additional unlisted clause, which states that no action of higher priority is enabled. For example, if the priority of an action is 3, it is not enabled if an action of priority 1 or 2 is enabled.

The fourth column of Table 1 lists the *statement* of each action. If a process is enabled and executes an action, then the statement, which consists of a list of assignments of values to the process' local variables, is executed.

We follow Table 1 by a detailed explanation of each of the actions.

Explanation of the Actions of DISJ**.** We now give a detailed explanation of each of the actions of DISJ.

Action A1 **(Reset):** If a process x is invalid, it executes A1, and then becomes 0-valid. An invalid process cannot change its parent or its level without first executing A1.

Table 1: Actions of DISJ

A1 priority 1	Reset	$\neg\,Valid(x)$	$\longrightarrow$	$x.out \leftarrow 0$ $x.level \leftarrow Level(x)$ $x.parent \leftarrow \bot$
A2 priority 2	Finish	$x.out = 1$ $x.done \neq Done(x)$	$\longrightarrow$	$x.done \leftarrow Done(x)$
A3 priority 3	Initialize	$Input(x) = 1$ $0\_Valid(x)$ $\forall y \in N(x) : y.parent \neq x$	$\longrightarrow$	$x.out \leftarrow 1$ $x.color \leftarrow 1$ $x.level \leftarrow 0$ $x.done \leftarrow$ FALSE
A4 priority 3	Join y	$y \in N(x)$ $Input(x) = 0$ $0\_Valid(x)$ $y.out = 1$ $y.level + 1 = x.level$ $y.color = 1$ $\forall z \in N(x) : z.parent \neq x$	$\longrightarrow$	$x.parent \leftarrow y$ $x.out \leftarrow 1$ $x.color \leftarrow 0$ $x.done \leftarrow$ FALSE
A5 priority 3	Reverse Color	$1\_Valid(x)$ $Input(x) = 0$ $\neg\,Can\_Recruit(x) \vee (x.color = 0)$ $x.parent.color = x.color$ $\forall y \in Chldrn(x) : y.color \neq x.color$	$\longrightarrow$	$x.color \leftarrow \neg x.color$
A6 priority 3	Absorb Color	$1\_Valid(x)$ $Input(x) = 1$ $\neg\,Can\_Recruit(x) \vee (x.color = 0)$ $\forall y \in Chldrn(x) : y.color \neq x.color$ $\neg x.done$	$\longrightarrow$	$x.color \leftarrow \neg x.color$

Action A2 **(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 TRUE. Alternatively, if $x \in O_1$ can determine that the local BFS trees are not finished, $x.done$ is changed to FALSE. Both changes are accomplished by the execution of Action A2.

When construction of local BFS trees is finished, all the *done* variables change to TRUE in a convergecast wave beginning at the leaves. When that wave reaches $x \in I_1$, then x can no longer execute Action A6, causing *color lock* to percolate down its tree. When that happens with every tree, all executions of Action A5 cease, and the configuration is final.

Action A3 **(Initialize):** If a process $x \in I_1$ is 0-valid, it initiates a local BFS tree with itself as the root, unless there is some neighbor y such that $y.parent = x$. The reason for this clause is that, otherwise, y could accidentally and erroneously link with the local BFS tree.

If such a y exists, then y is invalid, which implies that it will execute Action A1 during the next round, after which x is enabled to execute A3.

Action A4 **(Join):** If a process $x \in I_0$ is 0-valid and has a neighbor $y \in O_1$, and if $y.color = 1$ and $x.level = 1 + y.level$, then x can join y by executing Action A4, unless there is some neighbor z such that $z.parent = x$. The reason for this clause is the same as the reason given for Action A3.

When x joins y, $x.color \leftarrow 0$. Thus, x starts a 0-color wave, which follows the 1-color wave that y belongs to.

Action A5 **(Reverse Color):** Color waves alternate in color, and no color wave can pass its 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 $x.parent$.

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 $y = x.parent$. 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 process $x \in I_1$ 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.

If $x \in I_1$ and $x.done =$ TRUE, the local tree is complete, and color locking is desired. In this case, x refuses to absorb its current color wave, the color waves "pile up" behind it, and color lock is achieved. When all local BFS trees reach color lock, the configuration of DISJ is final, and $x.out = 1$ for all x.

3.2 Legitimate Configurations

There are two kinds of legitimate configurations. We say that a process x is in a *legitimate state of type 0* if the following conditions hold.

1. $\mathit{Input}(x) = 0$.
2. $0\_\mathit{Valid}(x)$.
3. $N(x) \subseteq O_0$.

We say that a process x is in a *legitimate state of type 1* if the following conditions hold.

1. $1\_\mathit{Valid}(x)$.
2. $x.\mathit{done}$.
3. $y.\mathit{done}$ for all $y \in \mathit{Chldrn}(x)$.
4. If $\mathit{Input}(x) = 0$, then $x.\mathit{parent}.\mathit{color} \neq x.\mathit{color}$.

We say that a configuration is *legitimate*, of type 0 or 1, if all processes are in a legitimate state of type 0 or 1, respectively.

Properties of Legitimate Configurations. If the configuration is legitimate of type 0, all processes have the same state, where $\mathit{level} = \infty$. If the configuration is legitimate of type 1, the network is partitioned into clusters, each of which contains exactly one member of I_1. Each process belongs to the cluster containing the nearest member of I_1 (where ties are broken arbitrarily), and the *parent* pointers of the processes of each cluster form a BFS tree rooted at its member of I_1.

4 Self-Stabilization and Silence

Our main result is Theorem 4.19 below, which follows immediately from the lemmas proved in this section.

4.1 Legitimacy and Silence

Remark 4.1 *Every legitimate configuration is final.*

Proof. Assume that the configuration is legitimate, and let x be a process.

Since $\mathit{Valid}(x)$, x cannot execute Action A1.

Suppose x is legitimate of type 0.

Since $x \in O_0$, x cannot execute Action A2.
Since $\mathit{Input}(x) = 0$, x cannot execute Action A3.
Since $N(x) \subseteq O_0$, x cannot execute Action A4.
Since $x \in O_0$, $1\_\mathit{Valid}(x) = \text{FALSE}$, and thus x cannot execute either Action A5 or A6.

Suppose x is legitimate of type 1.
For any $y \in N(x)$, since y is legitimate, $y \in O_1$ and $y.\mathit{done} = \text{TRUE}$. Thus, $\mathit{Done}(x) = \text{TRUE}$, and hence x cannot execute Action A2.

Since $x \in O_1$, x cannot execute either Action A3 or A4.
If $x \in I_0$, then $x.\mathit{parent}$ is legitimate of type 1. Then x cannot execute Action A5 since $x.\mathit{parent}.\mathit{color} \neq x.\mathit{color}$.
If $x. \in I_1$, then x cannot execute Action A6 since $x.\mathit{done} = \text{TRUE}$.

Thus, in either case, x is not enabled. and we are done.

We now prove the converse of Remark 4.1.

Lemma 4.2 *Every final configuration is legitimate.*

Proof. Assume that the current configuration of DISJ is final, but not legitimate. For any process x, we have $\mathit{Valid}(x) = \text{TRUE}$, and $x.\mathit{done} = \mathit{Done}(x)$ if $x \in O_1$, since otherwise x would be enabled to execute either Action A1 or A2.

Our proof is by contradiction. Assume that not all processes are in a legitimate state.

Case I: There is some $x \in O_0$ where x is not in a legitimate state, and $N(x) \subseteq O_0$. Then $0\_\mathit{Valid}(x)$. Since x is not legitimate of type 0, $\mathit{Input}(x) = 1$. Since all neighbors of x are valid, x is enabled to execute Action A3, contradiction.

Case II: There is some $x \in O_1$ such that $x.\mathit{level} > 0$ and $x.\mathit{parent}.\mathit{color} = x.\mathit{color}$. Without loss of generality, the level of x is maximum, *i.e.,* $y.\mathit{parent}.\mathit{color} \neq y.\mathit{color}$ for all $y \in O_1$ such that $y.\mathit{level} > x.\mathit{level}$.

If $\mathit{Can\_Recruit}(x) = \text{FALSE}$, then x is enabled to execute Action A5, since $y.\mathit{color} \neq x.\mathit{color}$ for all $y \in \mathit{Chldrn}(x)$. Suppose $\mathit{Can\_Recruit}(x) = \text{TRUE}$. Then there exists $y \in N(x) \cap O_0$ such that $y.\mathit{level} = 1 + x.\mathit{level}$, and $\mathit{Level}(y) = 1 + x.\mathit{level}$ since y is legitimate. Thus, y is enabled to execute Action A4. In either case, we have a contradiction.

Case III: There are processes $x \in O_0$ and $y \in N(x) \cap O_1$, and $z.\mathit{color} \neq z.\mathit{parent}.\mathit{color}$ for all $z \in O_1$ such that $z.\mathit{level} > 0$. Let r be the end of the chain starting with y and following *parent* pointers. Then $r \in O_1$ and $r.\mathit{level} = 0$. Since $y.\mathit{done} = \text{FALSE}$, it follows by induction along the chain that $r.\mathit{done} = \text{FALSE}$. If $r \in B_1$ and $\mathit{Can\_Recruit}(r)$, then some neighbor of r can execute Action A4, contradiction. Otherwise, r is enabled to execute Action A6, contradiction.

4.2 Characteristics of a Legitimate Configuration

Lemma 4.3 *In a legitimate configuration, $x.level = L(x)$ for all x.*

Proof. By Remark 4.1, the configuration is final. If $Output = 0$, then $Level(x) = \infty$, since otherwise Action A1 would be enabled.

Suppose $Output = 1$, and $x.level \neq L(x)$ Without loss of generality, $L(x)$ is minimum subject to that condition. If $x \in I_1$, then x is enabled to execute Action A1, contradiction. Henceforth, assume $x \in I_0$, which implies that $x.level \neq L(x) => 0$.

Case I: $x.level > L(x)$. Pick $r \in I_1$ such that $||r, x|| = L(x)$. Pick $y \in N(x)$ on the shortest path from x to r. Then $Level(x) \leq 1 + y.level = 1 + L(y) = L(x) < x.level$. Thus, x is enabled to execute Action A1, contradiction.

Case II: $x.level < L(x)$. For all $y \in N(x)$, $L(y) \geq L(x) - 1$, by the triangle inequality. Thus $Level(x) \geq L(x) > x.level$, which implies that x is enabled to execute Action A1, contradiction.

Corollary 4.4 *In a configuration which is legitimate of type 1, the network is partitioned into clusters, each containing one member of I_1. In each cluster, the parent pointers form a BFS tree rooted at its member of I_1.*

4.3 Energy

At any configuration of DISJ, and for any process x, let

$$Energy(x) = \begin{cases} 0 \text{ if } x.out = 0 \\ 1 \text{ if } (x.out = 1) \wedge (Chldrn(x) = \emptyset) \wedge (x.color = 0) \\ 2 \text{ if } (x.out = 1) \wedge (Chldrn(x) = \emptyset) \wedge (x.color = 1) \\ \max\Big(\{1 + Energy(y) : (y \in Chldrn(x)) \wedge (y.color \neq x.color)\} \cup \\ \quad \{2 + Energy(y) : (y \in Chldrn(x)) \wedge (y.color = x.color)\}\Big) \text{ otherwise} \end{cases}$$

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

Lemma 4.5 *Execution of Action* A1*,* A2*,* A4*, or* A5 *by any process does not increase Max_Energy.*

Proof. If a process x executes Action A1, there is no effect on *Max_Energy* if $x \in O_0$. If $x \in O_1$, then $Energy(x) \leftarrow 0$, and the effect on the energy of any process is non-positive. If x executes Action A2, there is no effect on color, and thus *Max_Energy* is unaffected.

Suppose x executes Action A4, attaching itself to $y \in N(x)$. Then $\mathit{Energy}(x) \leftarrow 1$, but $\mathit{Energy}(y) = 2$ before the step. Thus, *Max_Energy* does not decrease.

Suppose x executes Action A5. Let $y = x.\mathit{parent}$. Before the step, $\mathit{Energy}(y) \geq \mathit{Energy}(x) + 2$. Thus, the action could increase the energy of x by at most one.

Energy Plus Level. For any process x, define

$$\mathit{Energy\_plus\_level}(x) = \begin{cases} 0 \text{ if } x.\mathit{level} = \infty \\ \mathit{Energy}(x) + x.\mathit{level} \text{ otherwise} \end{cases}$$

Define *Max_Energy_plus_level* to be the maximum value of $\mathit{Energy\_plus\_level}(x)$ over all x.

Lemma 4.6 *The value of* $\max \left\{ \begin{matrix} \mathit{Max\_Energy\_plus\_level} \\ 2n \end{matrix} \right\}$ *cannot increase.*

Proof. If x is not a root, then $\mathit{Energy\_plus\_level}(x) \leq \mathit{Energy\_plus\_level}(x.\mathit{parent})$ by the definition of *Energy*. Thus, the maximum value of *Energy_plus_level*, if greater than zero, is always achieved at a root, either a true root or a false root.

Let $\gamma \mapsto \gamma'$ be a step, and let M, M' be the values of $\max\{2n, \mathit{Max\_Energy\_plus\_level}\}$ at γ and γ', respectively. We use "prime" notation for the values of variables and functions at γ', and no 'prime' to indicate values at γ.

We need to prove $M' \leq M$. If $M' \leq 2n$, we are done. Therefore, we can assume that $M' > 2n$. Pick x such that $\mathit{Energy\_plus\_level}'(x) = M'$. If x is a true root at γ', then $\mathit{Energy\_plus\_level}'(x) = \mathit{Energy}'(x) \leq 2n$. Thus, x is a false root at γ'.

If x did not execute at the step, then $\mathit{Energy}'(x) \leq \mathit{Energy}(x)$ and thus $M' = \mathit{Energy\_plus\_level}'(x) \leq \mathit{Energy\_plus\_level}(x) \leq M$. If x executed at the step, then x could not have been a false root at γ, Let $y = x.\mathit{parent}$. Then $\mathit{Energy}'(x) < \mathit{Energy}(y)$ by the definition of *Energy*. Since $x.\mathit{level} = 1 + y.\mathit{level}$, we have $M' = \mathit{Energy\_plus\_level}'(x) \leq \mathit{Energy\_plus\_level}(y) \leq M$.

4.4 Silence

We define an infinite computation of an algorithm to be *repetitive* if every configuration that occurs in the computation occurs infinitely often, and if every transition between two configurations that occurs also occurs infinitely often.

Lemma 4.7 *If* DISJ *has an infinite computation, then* DISJ *has a repetitive infinite computation.*

Proof. Let Γ be an infinite computation of DISJ on a network $\mathcal{G}$. Let M be the value of *Max_Energy_plus_level* at the first configuration of Γ. By Lemma

4.6, $\mathit{Max\_Energy\_plus\_level} \leq M$ at all configuration of Γ. Thus, the number of possible values of $x.\mathit{level}$ for any given process $x \in \mathcal{G}$. is bounded by $M + 1$. The number of possible values of $x.\mathit{parent}$ is bounded by the degree of the network, and the number of possible values of every other variable of DISJ is bounded as well. Thus, the number of distinct configurations in the computation Γ is finite.

Let $\mathcal{P}$ be the set of distinct consecutive pairs of configurations of DISJ which occur in the computation $\mathcal{G}$.

Let $\mathcal{F}$ be the set of members of $\mathcal{P}$ that occur only finitely many times in Γ. Since $\mathcal{F}$ is finite, there is some step γ of Γ after which no member of $\mathcal{F}$ occurs. Let Γ' be obtained by deleting all steps of Γ up to and including γ. In Γ', every pair of consecutive configurations is repeated infinitely many times, and thus Γ' is repetitive.

We now prove that DISJ is silent. Our proof is by contradiction – throughout the remainder of this subsection, we assume that Γ is an infinite computation of DISJ. Without loss of generality, by Lemma 4.7, Γ is repetitive.

Sets of Processes.

1. S = the set of processes which never execute.
2. A = the set of processes which execute.
3. EO_i, for $i = 0, 1$, is the set of processes which are in O_i forever.
4. $EB = EO_0 \cup EO_1$.
5. EO_1C_j, for $j = 0, 1$, is the set of processes in EO_1 whose color remains j forever.
6. $EO_1C = EO_1C_0 \cup EO_1C_1$.

Remark 4.8 *If $x \in EB$, then $x.\mathit{level}$ cannot change.*

Lemma 4.9 *If $\mathit{Input}(x) = 1$, then $x \in EB$.*

Proof. If $x \in S$, then $x \in EB$. If $y \in N(x) \cap S$, and $y.\mathit{parent} = x$. Then x cannot execute either Action A3 or A4, and hence $x \in EB$. Otherwise, suppose $x \notin EB$. Then x will eventually execute Action A3, and will never again execute Action A1, hence $x \in EO_1$, contradiction.

Define the function f on processes.

$$f(x) = \begin{cases} \infty \text{ if } x \in EO_0 \\ x.\mathit{level} \text{ if } x \in EO_1 \\ 1 + \min\{f(y) : y \in N(x)\} \text{ otherwise} \end{cases}$$

Lemma 4.10 *$x.\mathit{level} \geq f(x)$.*

Proof. By contradiction. Let $\Lambda = \min\{x.level : x.level < f(x)\}$. If $x.level = \Lambda$ and $x.level < f(x)$, then x will execute Action A1. When all such processes have executed, Λ will increase. Since Λ is bounded above by the diameter of the network, eventually $x.level \geq f(x)$.

Lemma 4.11 *All processes are in EB.*

Proof. By contradiction. Suppose $x \notin EB$. Let h be the minimum value of $x.level$, taken over all configurations of Γ. If $h = 0$, then $Input(x) = 1$, and hence $x \in EB$ by Lemma 4.9. Otherwise, there is some $y \in N(x)$ such that $y \in EO_1$ and $y.level = h - 1$. Thus, $Level(x) \leq h$ at every configuration of Γ. Since any neighbor of x whose level is less than h must be in EO_1, we have that $Level(x)$ cannot change, and hence must always be equal to h. Thus, x will remain valid and cannot execute Action A1. Hence $x \in EO_1$, contradiction.

Corollary 4.12 *For any process x, x.parent never changes.*

Lemma 4.13 $EO_0 \subseteq S$.

Proof. If $x \in EO_0$, then the only action that x could execute is A1. By Remark 4.8 and Lemma 4.11, no valid process in EO_0 can become invalid, and thus $Level(x)$ cannot change. Thus, x can execute Action A1 at most once.

Lemma 4.14 *if $x \in EO_1$ and either $x.parent \in EO_1C$ or $y \in EO_1C$ for some $y \in Chldrn(x)$, then $x \in EO_1C$.*

Proof. By the guards of Actions A5 and A6, x cannot execute either of those actions more than once.

Lemma 4.15 $EO_1 \subseteq S$.

Proof. By contradiction. By Lemma 4.11 and Corollary 4.13, $A \subseteq EO_1$.

We first prove, by contradiction, that $x.done$ never changes for any $x \in A$. Let x be the process of greatest level such that $x.done$ changes. But $Done(x)$ cannot change, and so x can execute Action A2 at most once, contradiction.

Let $\mathcal{A}$ be the graph whose nodes are processes in A and whose edges are defined by the parent pointers. Each component of $\mathcal{A}$ is a tree rooted at its member of minimum level. Let $\mathcal{T}$ be one of those trees. If any member $x \in \mathcal{T}$ is a neighbor of any member of EO_0, then x can change color at most once. Thus, by Lemma 4.14 applied inductively, $\mathcal{T} \subseteq EO_1C$. If any member of $\mathcal{T}$ is linked, by a parent pointer, to any process not in A, then, also by Lemma 4.14 applied inductively, $\mathcal{T} \subseteq EO_1C$. Since no value of *done* in $\mathcal{T}$ can change, $\mathcal{T} \subseteq S$.

Now, suppose that no member of $\mathcal{T}$ is a neighbor of any member of EO_0 or is linked by a parent pointer to any process not in A. Then the root of $\mathcal{T}$ has level 0 and $x.done = \text{TRUE}$ for all $x \in \mathcal{T}$. The root of $\mathcal{T}$ cannot execute Action A6, and thus, by Lemma 4.14 applied inductively, $\mathcal{T} \subseteq EO_1C$, and thus $\mathcal{T} \subseteq S$.

4.5 Time Complexity of DISJ

Using the concept of energy, we can prove that, in the case that $Output = 0$, energy must decrease during every round, and thus must reach zero after at most $2n$ rounds. After one more round, a legitimate configuration of type 0 is achieved.

In the case that $Output = 1$, color waves actually slow down convergence. A simple flooding algorithm, which would work if we were guaranteed that $I_1 \neq \emptyset$, would take at most *Diam* rounds, where *Diam* is the diameter of $\mathcal{G}$. Unfortunately, the addition of color waves causes the case where $Output = 1$ to also take $O(n)$ rounds.

Time Complexity when $I_1 = \emptyset$

In this subsection, we analyze the time complexity of DISJ in the case that $I_1 = \emptyset$.

Lemma 4.16 *If $I_1 = \emptyset$ and $O_1 \neq \emptyset$, then Max_Energy decreases during the next round.*

Proof. During the first round, all the first processes of chains with energy *Max_Energy* will execute Action A1. The remaining chains will have smaller energy. Since $I_1 = \emptyset$, no process can execute Action A4. By Lemma 4.5, no other action can increase *Max_Energy*. Thus *Max_Energy* decreases.

Lemma 4.17 *If $I_1 = \emptyset$, then the configuration will be legitimate of type 0 within $2n + 1$ rounds of an arbitrary initialization.*

Proof. In the initial configuration, $Max\_Energy \leq 2n$, By Lemma 4.16, within $2n$ rounds, $Max\_Energy = 0$, and thus $O_1 = \emptyset$. Within one more round, every process which is not valid will execute Action A1, and the configuration will then be legitimate of type 0.

Lemma 4.18 *If $Output = 1$, then* DISJ *converges within $O(n)$ rounds.*

We only sketch the proof. The initial value of *Max_Energy* cannot exceed $2n$. We can show that within $O(n)$ rounds, $Max\_Energy = O(Diam)$, after which DISJ converges within $O(Diam)$ additional rounds.

From the above lemmas, we conclude our main result, below.

Theorem 4.19 DISJ *is self-stabilizing and silent, takes $O(n)$ rounds, and works under the unfair daemon.*

Proof. Let Γ be any computation of DISJ. By Lemmas 4.11 and 4.15, and Corollary 4.13, Γ is finite. By Lemma 4.2, the last configuration of Γ is legitimate, and hence DISJ is self-stabilizing and silent. By Lemmas 4.18 and 4.17, DISJ converges in $O(n)$ rounds from an arbitrary initial configuration.

References

1. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of Computing Machinery **17** (1974) 643–644
2. Dolev, S.: Self-Stabilization. The MIT Press (2000)
3. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing leader election in optimal space under an arbitrary scheduler. Theoretical Computer Science **412**(40) (2011) 5541–5561