Optimal Deterministic Self-stabilizing Vertex Coloring in Unidirectional Anonymous Networks

Samuel Bernard*, Stéphane Devismes†, Maria Gradinariu Potop-Butucaru*, and Sébastien Tixeuil*
*LIP6 - Université Pierre et Marie Curie - Paris, France
Email: {samuel.bernard,maria.gradinariu,sebastien.tixeuil}@lip6.fr
†VERIMAG - Université Joseph Fourier - Grenoble, France
Email: stephane.devismes@imag.fr

Abstract

A distributed algorithm is self-stabilizing if after faults and attacks hit the system and place it in some arbitrary global state, the systems recovers from this catastrophic situation without external intervention in finite time. Unidirectional networks preclude many common techniques in self-stabilization from being used, such as preserving local predicates. In this paper, we investigate the intrinsic complexity of achieving self-stabilization in unidirectional anonymous general networks, and focus on the classical vertex coloring problem. Specifically, we prove a lower bound of n states per process (where n is the network size) and a recovery time of at least $n(n-1)/2$ actions in total. We also provide a deterministic algorithm with matching upper bounds that performs in arbitrary unidirectional anonymous graphs.

1. Introduction

One of the most versatile technique to ensure forward recovery of distributed systems is that of *self-stabilization* [1], [2]. A distributed algorithm is self-stabilizing if after faults and attacks hit the system and place it in some arbitrary global state, the systems recovers from this catastrophic situation without external (*e.g.* human) intervention in finite time. Self-stabilization makes no hypotheses about the extent or the nature of the faults and attacks that may harm the system, yet may induce some overhead (*e.g.* memory, time) when there are no faults, compared to a classical (*i.e.* non-stabilizing) solution. Computing space and time bounds for particular problems in a self-stabilizing setting is thus crucial to evaluate the impact of adding forward recovery properties to the system.

The vast majority of self-stabilizing solutions in the literature [2] considers bidirectional communications capabilities, *i.e.* if a process u is able to send information to another process v, then v is always able to send information back to u. This hypothesis is valid in many cases, but cannot capture the fact that asymmetric situations may occur, *e.g.* in wireless networks [3], it is possible that u is able to send information to v yet v cannot send any information back to u (u may have a wider range antenna than v). Asymmetric situations, that we denote in the following under the term of *unidirectional* networks, preclude many common techniques in self-stabilization from being used, such as preserving local predicates (a process u may take an action that violates a predicate involving its outgoing neighbors without u knowing it, since u cannot get any input from its outgoing neighbors).

This work was supported by ANR SHAMAN project.

1.1. Related Works

Self-stabilization in bidirectional networks makes a distinction between *global* tasks (*i.e.* tasks whose specification forbids particular state combinations of processes arbitrarily far from one another in the network, such as leader election) and *local* tasks (whose specification forbids particular state combinations only for neighboring processes, such as vertex coloring). Self-stabilizing solutions for local tasks are often considered easier in bidirectional networks since detecting incorrect situations requires less memory and computing power [4], recovering can be done locally [5], and strict Byzantine containment can be guaranteed [6], [7].

Since a self-stabilizing algorithm may start from any arbitrary state, lower bounds for non-stabilizing (*a.k.a.* properly initialized) distributed algorithms still hold for self-stabilizing ones. As a result, relatively few works investigate lower bounds that are specific to self-stabilization [8], [9], [10], [11], [12], [13]. Results related to space lower bounds deal with global tasks (*e.g.* constructing a spanning tree [9], finding a center [9], electing a leader [8], [9], passing a token [10], [11], [13], etc.). [12] provides a time lower bound for self-stabilizing token passing, still a global task. Global tasks typically require $\Omega(n)$ states per process (*i.e.* $\Omega(\log(n))$ bits per process) and $\Omega(n)$ time complexity to recover from faults.

Investigating the possibility of self-stabilization in unidirectional networks was recently emphasized in several

papers [14], [15], [16], [17], [18], [19], [20][1]. In particular, [16] shows that in the simple case of acyclic unidirectional networks, nearly any recursive function can be computed anonymously in a self-stabilizing way. Computing global tasks in a general topology requires either unique identifiers [14], [15], [18] or distinguished processes [17], [19], [20]. Observe that all aforementioned works consider global tasks, and provide constructive upper bound results (*i.e* algorithms), leaving the question of matching lower bounds open.

1.2. Our Contribution

In this paper, we investigate the intrinsic complexity of achieving self-stabilization in unidirectional anonymous uniform networks. In such networks, processes do *not* have identifiers, and no process (or subset of processes) is distinguished. We focus on the classical vertex coloring problem, a local task that can be self-stabilizingly solved in bidirectional networks with a memory requirement and a number of moves per process that only depend on the degree of the communication graph [21], [22]. In bidirectional networks, deterministic solutions require only a number of states that is proportional to the network maximum degree Δ, and the number of actions per process in order to recover is $O(\Delta)$. Moreover, since the length of the chain of causality after a correcting action is executed is only one, strict Byzantine containement can be achieved [7].

To satisfy the vertex coloring specification in unidirectional general networks, an algorithm must ensure that no two neighboring nodes (*i.e.* two nodes u and v such that either u can send information to v, or v can send information to u, but not necessarily both) have identical colors. The main result of this paper is to show that solving vertex coloring in unidirectional anonymous general networks with a deterministic algorithm is much harder than solving the same task in bidirectional networks. Specifically, we prove a lower bound of n states per process (where n is the network size) and a recovery time of at least $n(n-1)/2$ actions in total (and thus $\Omega(n)$ actions per process). This result essentially implies that scalability of tasks in unidirectional networks induces a complexity cost proportional to the size of the network while in bidirectional networks it may depend on other parameters (i.e. network degree) that can be maintained constant even if the system size changes. Moreover, we present a deterministic algorithm for vertex coloring in unidirectional networks with matching upper bounds that performs in arbitrary graphs. The protocol is thus optimal for both space and time complexity.

1. Please refer to [2] for additional references

1.3. Outline

The remaining of the paper is organized as follows: Section 2 presents the programming model and problem specification, Section 3 provides impossibility results and lower bounds for the vertex coloring problem in unidirectional anonymous general networks, while Section 4 presents a deterministic algorithm matching the upper bounds. Section 5 gives some concluding remarks and open questions.

2. Model

2.1. Distributed Program Model

A distributed program consists of a set V of n processes which may not have unique identifiers. Therefore, processes will be referred in the following as anonymous. A process maintains a set of variables that it can read or update, that define its *state*. Each variable ranges over a fixed domain of values. We use small case letters to denote singleton variables, and capital ones to denote sets. A process contains a set of *constants* that it can read but not update. A set E of oriented edges is defined as follows: $(i,j) \in E$ if and only if j can read the variables maintained by i. In this case, i is called a *predecessor* of j, and j is called a *successor* of i. The set of predecessors (resp. successors) of i is denoted by $P.i$ (resp. $S.i$), and the union of predecessors and successors of i is denoted by $N.i$, the *neighbors* of i. In some case, we are interested in the iterated notions of those sets, *e.g.* $S.i^0 = \{i\}$, $S.i^1 = S.i$, and $S.i^k = \cup_{j \in S.i} S.j^{k-1}$. The values $\delta_{in}.i$, $\delta_{out}.i$, and $\delta.i$ denote respectively $|P.i|$, $|S.i|$, and $|N.i|$; Δ_{in}, Δ_{out}, and Δ denote the maximum possible values of $\delta_{in}.i$, $\delta_{out}.i$, and $\delta.i$ over all processes in V.

An action has the form $\langle guard \rangle \longrightarrow \langle command \rangle$. A *guard* is a Boolean predicate over the variables of the process and its predecessors. A *command* is a sequence of statements assigning new values to the variables of the process. We refer to a variable v of process i as $v.i$. A *parameter* is used to define a set of actions as one parameterized action.

A *configuration* of the distributed program is the assignment of a value to every variable of each process from its corresponding domain. Each process contains a set of actions referred in the following as *algorithm*. In the following we consider that processes are *uniform*. That is, all the processes contain the exact same set of actions. An action is *enabled* in some configuration if its guard is **true** in this configuration. A *computation* is a maximal sequence of configurations such that for each configuration γ_i, the next configuration γ_{i+1} is obtained by executing the command of at least one action that is enabled in γ_i. Maximality of a computation means that the computation is infinite or eventually reaches a *terminal configuration* (*i.e.*, a configuration where no action is enabled).

A configuration *conforms* to a predicate if this predicate is **true** in this configuration; otherwise the configuration *violates* the predicate. By this definition every configuration conforms to predicate **true** and none conforms to **false**. Let R and S be predicates over the configurations of the program. Predicate R is *closed* with respect to the program actions if every configuration of the computation that starts in a configuration conforming to R also conforms to R. Predicate R *converges* to S if R and S are closed and any computation starting from a configuration conforming to R contains a configuration conforming to S. The program *deterministically stabilizes* to R if and only if **true** converges to R.

A *scheduler* is a predicate on computations, that is, a scheduler define a set of possible computations, such that every computation in this set satisfies the scheduler predicate. We distinguish two particular schedulers in the sequel of the paper: the *distributed* scheduler corresponds to predicate **true** (that is, all computations are allowed); in contrast, the *locally central* scheduler implies that from any configuration belonging to a computation satisfying the scheduler, no two enabled actions are executed simultaneously on neighboring processes.

2.2. Problem Specification

Consider a set of colors ranging from 0 to $\mathtt{k}-1$, for some integer $\mathtt{k} > 1$. Each process i defines a function $color.i$ that takes as input the states of i and its predecessors, and outputs a value in $\{0, \ldots, \mathtt{k}-1\}$. The *unidirectional vertex coloring* predicate is satisfied if and only if for every $(i, j) \in E$, $color.i \neq color.j$.

3. Impossibility Results and Lower Bounds

This section is dedicated to impossibility results and lower bounds for *deterministic* vertex coloring programs that operate in every topology. Theorem 1 (presented below) shows that if unconstrained schedules (*i.e.* the scheduler is distributed) are allowed, there are some topologies for which initial symmetric configurations cannot be broken afterwards, making the unidirectional coloring problem impossible to solve by a deterministic algorithm. This justifies the later hypothesis of a locally central scheduler in Section 4, *i.e.* a scheduler that never schedules for execution two neighboring enabled processes *simultaneously*.

Lemma 1 *Let us consider an unidirectional anonymous ring network $\{p_0, p_1, \ldots, p_{n-1}\}$ of size n. Consider every node executes a uniform deterministic self-stabilizing vertex coloring algorithm. If there exists a configuration c such that, for every $i \in \{0, \ldots, n-1\}$, $s.p_i = s.p_{(i-1) \bmod n}$ holds, the coloring specification is not satisfied in c.*

Proof: By hypothesis, the color of a node depends only on its state and that of its predecessors. In a ring network, the color of a node depends only of its own state and that of its predecessor. Since all nodes have the same state and the same predecessor's state, all nodes have the same color. As a result, two neighboring nodes have the same color, and the coloring problem is not satisfied. □

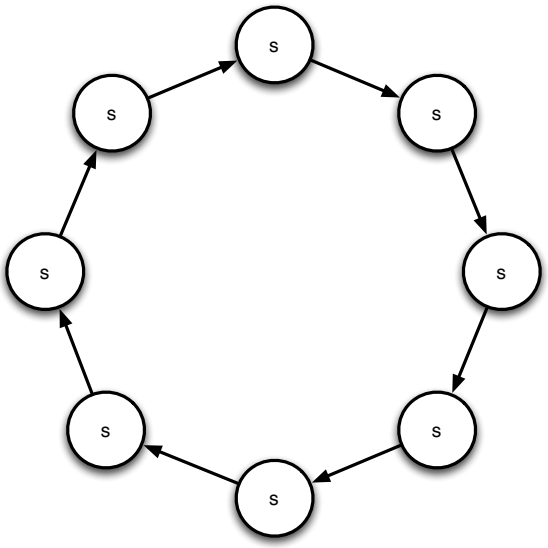
Lemma 2 *Let us consider an unidirectional anonymous ring network $\{p_0, p_1, \ldots, p_{n-1}\}$ of size n. Consider every node executes a uniform deterministic self-stabilizing vertex coloring algorithm. Whenever there exists i such that $s.p_i = s.p_{(i-1) \bmod n}$, p_i is enabled and if activated would change its state to $s'.p_i$ with $s'.p_i \neq s.p_i$.*

Proof: We first show that p_i is enabled. Assume the contrary, and consider a uniform ring of n nodes that are all in the same state. If p_i is not enabled, none of the remaining processes is enabled either. Hence, the configuration is terminal. Now, a process p_i may only read its own state and that of its predecessor in the ring, so the color $color.p_i$ is uniquely determined by these two values only. Moreover, the $color.p_i$ is the same as $color.p_{(i-1) \bmod n}$. So, the configuration is terminal and two neighboring processes have the same color. This contradicts the fact that the algorithm is a deterministic self-stabilizing unidirectional vertex coloring one.

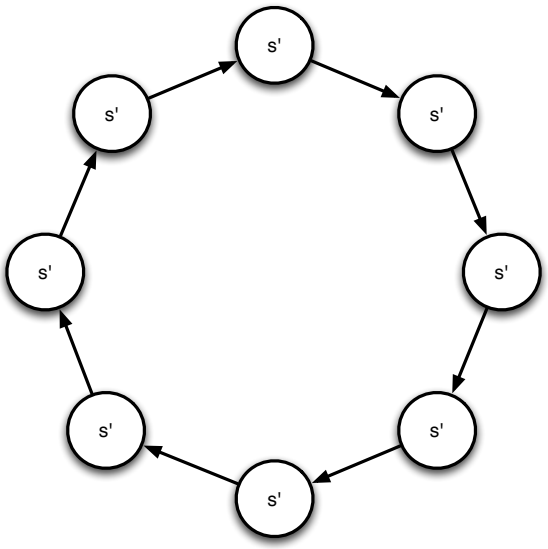
Then we show that p_i, if activated moves to a different state $s'.p_i$. Assume p_i moves to the same state $s.p_i$, then if the starting configuration is such that all nodes have the same state, then no node is able to change its state, the algorithm being uniform. Since this configuration cannot be a vertex coloring (Lemma 1), the system never changes the global configuration and thus is not self-stabilizing. □

Theorem 1 *There exists no uniform deterministic self-stabilizing vertex coloring algorithm that can run on arbitrary unidirectional graphs under a distributed scheduler.*

Proof: Assume, by the contradiction, that there exists a uniform deterministic self-stabilizing vertex coloring algorithm that runs on arbitrary unidirectional graphs under a distributed scheduler. Consider a unidirectional ring $\{p_0, p_1, \ldots, p_{n-1}\}$ of size n (the algorithm must in particular self-stabilizes in this topology). Assume that in the initial state all nodes are in the same state s (see Figure 1.(a)). By Lemma 2, all nodes are enabled, and if a node is activated by the scheduler, it moves to a different state s'. Consider the synchronous scheduler (a particular case of the distributed scheduler) that, at each step, activates all nodes. Then, after one scheduler activation, all nodes have state s' (see Figure 1.(b)). After another activation, all nodes move to state s'', etc. In this infinite execution, every configuration has all nodes with the same state and thus, the vertex coloring problem is not solved. As a result, the algorithm cannot be self-stabilizing. □



(a) uniform configuration in state s



(b) uniform configuration in state s'

Figure 1. A possible execution with synchronous scheduling

Notice that the result of Theorem 1 holds even if participating processes have infinite number of states.

From now on, we assume the scheduler is locally central.[2] We demonstrate that a uniform deterministic self-stabilizing algorithm for the unidirectional vertex coloring problem that can perform in arbitrary networks must use at least n states per process. As previously, the proof is by exhibiting a particular family of networks (namely, n-sized rings) in which the bound is reached by any such algorithm even assuming a locally central scheduler.

2. Note that in the model considered in the current paper the local central scheduler and central scheduler are equivalent.

Algorithm 1 A uniform deterministic vertex coloring algorithm for unidirectional rings

process i
const
$\mathtt{k}$: integer
$p.i$: predecessor of i
var
$c.i$: color of node i
action
$c.i = c.p.i \rightarrow$
$c.i := c.i + 1 \mod \mathtt{k}$

Lemma 3 *Assume every process executes a uniform deterministic self-stabilizing unidirectional vertex coloring algorithm that uses a finite number of states K and can perform in arbitrary networks of size n (possibly, K is related to n). There exists some networks and initial configurations such that the state sequences of every process (starting from those configurations) are isomorphic to that of Algorithm 1, for some parameter $k \leq K$.*

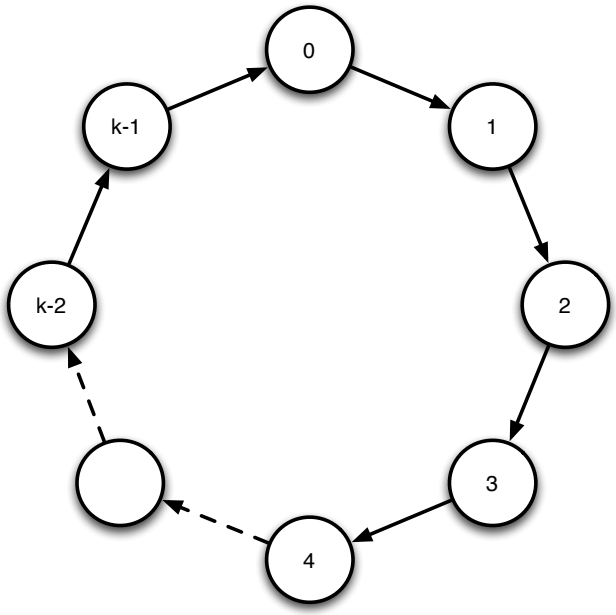


Figure 2. State transition function of Algorithm 1

Proof: Since the protocol is assumed to perform in arbitrary networks, it must also performs properly in unidirectional rings $\{p_0, p_1, \ldots, p_{n-1}\}$ of size n. Assume that in every execution, the (locally central) scheduler activates a node p_i only when its state $s.p_i$ is equal to $s.p_{(i-1) \bmod n}$,

the state of the predecessor of p_i in the ring. Then, the transition function of node p_i is solely based on the state $s.p_i$.

Let $s_0.i, s_1.i, \ldots$ be the sequence of states returned by the transition function of process p_i executing the self-stabilizing vertex coloring algorithm started in some state $s_0.i$. Since (*i*) the overall number of states of p_i s finite (it is bounded by K by hypothesis), (*ii*) p_i is enabled whenever $s.i = s.p.i$ (by Lemma 2), and (*iii*) the protocol is deterministic (by hypothesis), the sequence of states $s_0.i, s_1.i, \ldots$ is infinite and contains a recurring pattern of consecutive states. That is, there exists j and l ($j < l$) such that $s_j.i = s_l.i$ and $l - j \leq K$. Since we assume that the protocol is self-stabilizing, it may be started from any arbitrary state for every process, in particular this implies that every process p_i may be started from state $s_j.i$.

When all processes start in state s_j, and are activated only when they are in the same state as their predecessor, the only possible behavior is to move to the next state in the recurring pattern. That is, the transition function that is used by every process p_i is isomorphic to that of the same process executing Algorithm 1 (*e.g.* with isomorphism f such that $f(s_j) = j - i, \forall i \leq j < l$ and assuming $k = l - i$).

So, there exists a family of networks and initial configurations such that the state sequences of every process (starting from those configurations) are isomorphic to that of Algorithm 1, for some parameter $k \leq K$. □

Theorem 2 *Any uniform deterministic self-stabilizing protocol for unidirectional vertex coloring in general graphs requires at least n states per process where n is the size of the network.*

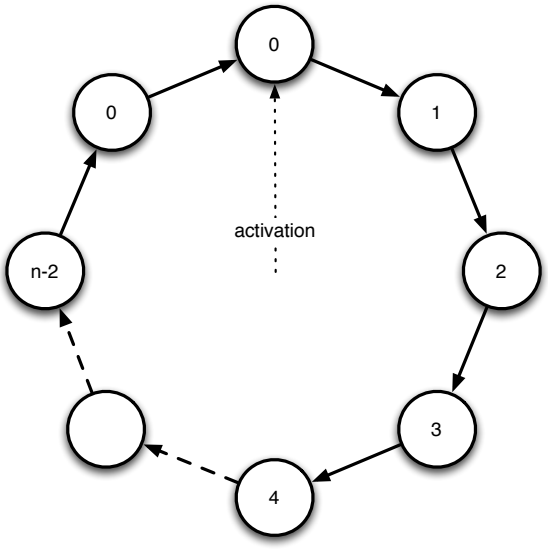


Figure 3. A possible execution of Algorithm 1: Starting configuration

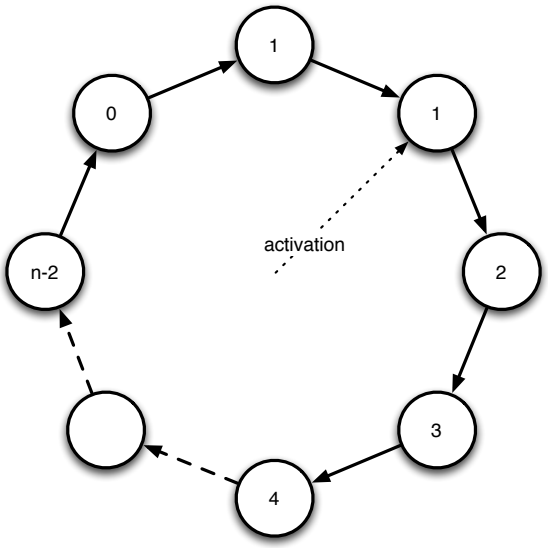


Figure 4. A possible execution of Algorithm 1: After 1 step

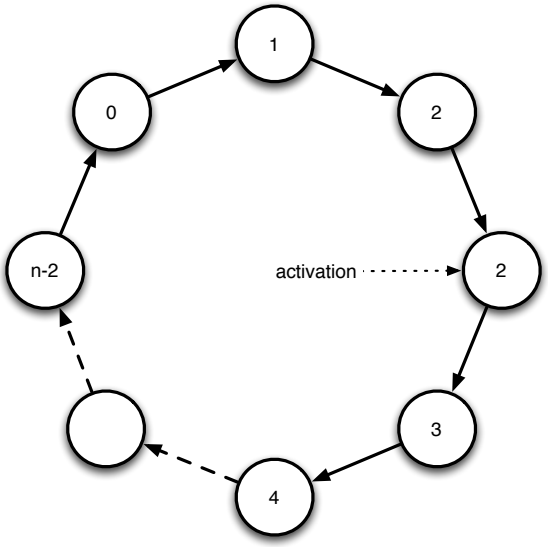


Figure 5. A possible execution of Algorithm 1: After 2 steps

Proof: Assume there exists a uniform deterministic self-stabilizing protocol for unidirectional vertex coloring in arbitrary graphs that requires (strictly) less than n states for a particular node. Since the protocol is uniform, every node must use $k < n$ states.

Since the protocol must perform in arbitrary n-sized graphs, it must also perform correctly in an n-sized unidirectional ring $\{p_0, p_1, \ldots, p_{n-1}\}$. In what follows, we consider executions of the protocol in which the scheduler only activates nodes that have the same state as their predecessor. By Lemma 3, the transition function of every node is

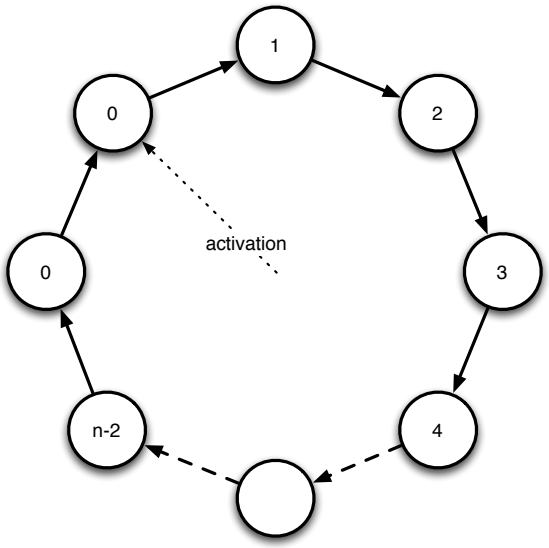


Figure 6. A possible execution of Algorithm 1: After $n-1$ steps

isomorphic to that of Algorithm 1, so we assume all nodes execute Algorithm 1 with $k < n$.

In the following, we consider $k = n - 1$ but the proof can be easily generalized for any $k < n$ by putting the $n-k+1$ last processors in the same state. Now consider the unidirectional ring presented in Figure 3. The scheduler only activates the single node with the same state 0 as its parent, and reach the configuration presented in Figure 4. The scheduler may now activate the single node with the same state 1 as its parent and reach the configuration presented in Figure 5. We repeat the argument and reach the configuration presented in Figure 6. This configuration is symmetric to that of the configuration presented in Figure 3, so the process can repeat infinitely often.

As a result, the protocol does not stabilize, and the initial hypothesis that there exists a deterministic self-stabilizing protocol that performs in every n-sized network is contradicted. □

We now address the question of time lower bounds for deterministic self-stabilizing programs for the unidirectional vertex coloring problem in general graphs.

Theorem 3 *Any uniform deterministic self-stabilizing protocol for unidirectional vertex coloring in arbitrary n-sized networks converges in at least $\frac{n(n-1)}{2}$ steps.*

Proof: Let us assume that there exists a deterministic self-stabilizing protocol for arbitrary unidirectional n-sized networks.

Since the protocol must perform in any arbitrary n-sized network, it must perform properly on a n-sized unidirectional chain, where processes are ordered from the sink p_1 to the source p_n.

Since the protocol is self-stabilizing, every process may be started in some arbitrary state s. We now consider a locally central scheduler that activates nodes according to the schedule presented in Schedule 1.

Schedule 1 Our $\frac{n(n-1)}{2}$-steps scheduling in n-sized chains

```
var
    i,j: integer
scheduler
    for j from n − 1 to 1
        for i from 1 to j
            activate p_i
```

Schedule 1 selects a single process at a time for execution, thus it satisfies the locally central scheduler property. In addition, it only selects for execution a process that has the same state as its predecessor (and that is thus enabled by Lemma 2): if $p_1, \ldots, p_k$ have the same state s then p_1 to p_{k-1} are enabled and if they are activated in ascending order, p_1 to p_{k-1} will move to the same "next" state s' (see Figure 2.(a)). So every process activation leads to an effective move and the total number of activations is $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.

By Lemma 1, it is necessary for the coloring specification to be satisfied, that a process does not have the same state as its predecessor, which implies that the total number of possible moves expressed above is actually the minimal number of moves before satisfying the coloring specification. □

4. Optimal Deterministic Unidirectional Vertex Coloring

In this section we propose a time and space optimal self-stabilizing deterministic algorithm for unidirectional coloring. The algorithm is referred thereafter as Algorithm 2 and performs under the locally central scheduler (see Theorem 1). The algorithm can be informally described as follows: each process i has an integer variable $c.i$ (that ranges from 0 to $\mathtt{k}-1$, where $\mathtt{k}$ is a parameter of the algorithm) that denotes its color; whenever a node has the same color as one of its predecessors, it changes its color to the next available color in the modulo-$\mathtt{k}$ ring (that is, a ring of nodes labelled from 0 to $\mathtt{k}-1$ and such that for every label i, node i is the successor of node $(i-1) \bmod \mathtt{k}$). Here, the $color.i$ function simply returns the color variable $c.i$ of i.

A configuration is *legitimate* if, for every process i, and for every predecessor $p \in P.i$, $c.i \neq c.p.i$. Obviously, a legitimate configuration satisfies the unidirectional coloring predicate (assuming $color.i$ return $c.i$) and is terminal (all guarded commands are disabled). There remains to show

Algorithm 2 A uniform deterministic coloring algorithm for general unidirectional networks

process i
const
$\mathtt{k}$: integer
$P.i$: set of predecessors of i
parameter
p : node in $P.i$
var
$c.i$: color of node i
action
$p \in P.i$, $c.i = c.p \rightarrow$
do $p \in P.i$, $c.i = c.p \rightarrow$
$c.i := c.i + 1 \mod \mathtt{k}$
od

Processors	P_3, P_2		P_{n-1}, P_1		...	P_{n-2}
States	0	1	2	3	...	$n-1$

Table 1. An example of color table

how fast the algorithm attains a legitimate configuration in the worst case for every possible locally central schedule.

Theorem 4 *Algorithm 2 is a (state-optimal) uniform deterministic self-stabilizing protocol for coloring nodes in unidirectional general networks of size n (when $\mathtt{k} = n$), assuming a locally central scheduler and converges in $\frac{n(n-1)}{2}$ steps to a legitimate configuration.*

Proof: Assume Algorithm 2 starts in an arbitrary initial configuration. We now consider the table that lists, for every possible color (in the set $\{0, \ldots, n-1\}$ since we assume $\mathtt{k} = n$), the processes that currently have this color. An example of such a table is presented as Table 1, where processes P_3 and P_2 have color 0, process P_{n-2} has color $n-1$, etc. This table is denoted in the sequel as the *color table*.

According to Algorithm 2 the evolution of the color table follows two rules:

1) A cell containing one process cannot become empty. That is, a process having a color not used by any other process in the system cannot be activated. In our algorithm, this is due to the fact that processes are enabled only if they share their color with their predecessor.
2) A process only moves to the right (in a cyclic manner) and cannot jump over an empty cell. Indeed, when activated, a process chooses the first "next" (in the sense of the usual total order on integers) unconflictual color hence the processes always move to the right. A process may move by several positions, but never skips a free position (this would mean that a process does not choose the "next" color although this color is not conflicting with any other process and thus not with the process predecessors).

Since there are n cells and n processes, every process could be placed in a different cell if necessary. Since a process cannot jump over an empty cell, after $n-1$ moves, a process is sure to find a free cell. In fact, the number of moves a process may have to perform to reach a free cell depends on the number of free cells. With k free cells, there are at most $n-k$ consecutive non-empty cells that could potentially provoke further conflicts. A process, in order to reach a free cell has to perform at most $n-k$ moves. Once this process occupies a free cell, the number of free cells decreases to $k-1$. Starting with $n-1$ free cell (every process has the same color), and finishing with 1, at most $1 + 2 + ... + (n-1) = \frac{n(n-1)}{2}$ steps are needed to have every process in a free cell or with a non-conflicting color (*i.e.* different from that of its predecessors) and thus reach a legitimate configuration. □

5. Conclusion

We investigated the intrinsic complexity of performing local tasks in unidirectional anonymous networks in a self-stabilizing setting. More specifically, we focus the vertex coloring problem. Contrary to "classical" bidirectional networks, local vertex coloring now induces global complexity (n states per process at least, n moves per process at least) for deterministic solutions. We presented a state and time optimal solution for the deterministic coloring. This work raises two important open questions:

1) Investigate the possibility of probabilistic coloring and its complexity.
2) The lower bounds we provide in the deterministic case rely on the existence of cycles in the unidirectional communication graph. We question the possibility of lowering those bounds (and matching upper bounds) for the vertex coloring problem in the case where the network is acyclic.

References

[1] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control." *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[2] S. Dolev, *Self-stabilization*. MIT Press, March 2000.

[3] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker, "Complex behavior at scale: An experimental study of low-power wireless sensor networks," UCLA Computer Science Department, Tech. Rep., 2002. [Online]. Available: http://citeseer.ist.psu.edu/533751.html

[4] J. Beauquier, S. Delaët, S. Dolev, and S. Tixeuil, "Transient fault detectors," *Distributed Computing*, vol. 20, no. 1, pp. 39–51, June 2007. [Online]. Available: http://www.springerlink.com/content/m267v22224127575/

[5] Y. Afek and S. Dolev, "Local stabilizer," *J. Parallel Distrib. Comput.*, vol. 62, no. 5, pp. 745–765, 2002.

[6] T. Masuzawa and S. Tixeuil, "Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults," *International Journal of Principles and Applications of Information Science and Technology (PAIST)*, vol. 1, no. 1, pp. 1–13, December 2007. [Online]. Available: http://210.119.33.7/paist/paper/2008_12/TOC2008_12.pdf

[7] M. Nesterenko and A. Arora, "Tolerance to unbounded byzantine faults," in *21st Symposium on Reliable Distributed Systems (SRDS 2002)*. IEEE Computer Society, 2002, pp. 22–29.

[8] J. Beauquier, M. Gradinariu, and C. Johnen, "Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings," *Distributed Computing*, vol. 20, no. 1, pp. 75–93, January 2007.

[9] S. Dolev, M. G. Gouda, and M. Schneider, "Memory requirements for silent stabilization," *Acta Inf.*, vol. 36, no. 6, pp. 447–462, 1999.

[10] S. Dolev, A. Israeli, and S. Moran, "Resource bounds for self-stabilizing message-driven protocols," *SIAM J. Comput.*, vol. 26, no. 1, pp. 273–290, 1997.

[11] P. Duchon, N. Hanusse, and S. Tixeuil, "Optimal randomized self-stabilizing mutual exclusion in synchronous rings," in *Proceedings of the International Conference on Distributed Computing (DISC 2004)*, ser. Lecture Notes in Computer Science, no. 3274. Amsterdam, The Nederlands: Springer Verlag, October 2004, pp. 216–229.

[12] C. Genolini and S. Tixeuil, "A lower bound on k-stabilization in asynchronous systems," in *Proceedings of IEEE 21st Symposium on Reliable Distributed Systems (SRDS'2002)*, Osaka, Japan, October 2002.

[13] S. Tixeuil, "On a space-optimal distributed traversal algorithm," in *WSS*, ser. Lecture Notes in Computer Science, A. K. Datta and T. Herman, Eds., vol. 2194. Springer, 2001, pp. 216–228.

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

[15] J. A. Cobb and M. G. Gouda, "Stabilization of routing in directed networks," in *WSS*, ser. Lecture Notes in Computer Science, A. K. Datta and T. Herman, Eds., vol. 2194. Springer, 2001, pp. 51–66. [Online]. Available: http://link.springer.de/link/service/series/0558/bibs/2194/21940051.htm

[16] S. Das, A. K. Datta, and S. Tixeuil, "Self-stabilizing algorithms in dag structured networks," *Parallel Processing Letters (PPL)*, vol. 9, no. 4, pp. 563–574, December 1999. [Online]. Available: http://dx.doi.org/10.1142/S0129626499000529

[17] S. Delaët, B. Ducourthial, and S. Tixeuil, "Self-stabilization with r-operators revisited," *Journal of Aerospace Computing, Information, and Communication (JACIC)*, vol. 3, no. 10, pp. 498–514, 2006. [Online]. Available: http://dx.doi.org/10.2514/1.19848

[18] S. Dolev and E. Schiller, "Self-stabilizing group communication in directed networks," *Acta Inf.*, vol. 40, no. 9, pp. 609–636, 2004.

[19] B. Ducourthial and S. Tixeuil, "Self-stabilization with r-operators," *Distributed Computing (DC)*, vol. 14, no. 3, pp. 147–162, July 2001. [Online]. Available: http://www.springerlink.com/content/0p4g0yt8vkd9jnlp/

[20] ——, "Self-stabilization with path algebra," *Theoretical Computer Science (TCS)*, vol. 293, no. 1, pp. 219–236, February 2003. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(02)00238-4

[21] M. Gradinariu and S. Tixeuil, "Self-stabilizing vertex coloring of arbitrary graphs," in *Proceedings of International Conference on Principles of Distributed Systems (OPODIS 2000)*, Paris, France, December 2000, pp. 55–70. [Online]. Available: http://hal.upmc.fr/hal-00631707/fr/

[22] N. Mitton, E. Fleury, I. Guérin-Lassous, B. Séricola, and S. Tixeuil, "On fast randomized colorings in sensor networks," in *Proceedings of ICPADS 2006*. IEEE Press, July 2006, pp. 31–38.

[23] A. K. Datta and T. Herman, Eds., *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2194. Springer, 2001.