Self-stabilizing Vertex Coloring of Arbitrary Graphs[*]

Maria Gradinariu[†] Sébastien Tixeuil[‡]

[†]IRISA, Campus de Beaulieu, Rennes, France
[‡]LRI-CNRS UMR 8623, Université Paris Sud, France.

Abstract

A self-stabilizing algorithm, regardless of the initial system state, converges in finite time to a set of states that satisfy a legitimacy predicate without the need for explicit exception handler of backward recovery. The vertex coloration problem consists in ensuring that every node in the system has a color that is different from any of its neighbors.

We provide three self-stabilizing solutions to the vertex coloration problem under unfair scheduling that are based on a greedy technique. We use at most $B + 1$ different colors (in complete graphs), where B is the node degree, and ensure stabilization within $O(n \times B)$ processor atomic steps. Two of our algorithms deal with uniform networks where nodes do not have identifiers. Our solutions lead to directed acyclic orientation and maximal independent set construction at no additional cost.

1 Introduction

1.1 Self-stabilization

Robustness is one of the most important requirements of modern distributed systems since various types of (transient) faults are likely to occur as these systems are exposed to constant change of their environment. One of the most inclusive approaches to fault-tolerance in distributed systems is *self-stabilization* [Dij74, Sch93]. Introduced by Dijkstra [Dij74], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior. Since most self-stabilizing fault-tolerant algorithms are nonterminating, if the distributed system is subject to transient faults corrupting the internal node state but not its behavior, once faults cease, the algorithms themselves guarantee to recover in a finite time to a safe state without any human intervention. This also means that the complicated task of initializing distributed systems is no longer needed, since self-stabilizing algorithms regain correct behavior regardless of the initial state. Furthermore, in practice, the context in which we may apply self-stabilizing algorithms is fairly broad since the program code can be stored in a stable storage at each node, so that it is always possible to reload the program after faults cease or after every fault detection.

1.2 Vertex Coloring

The vertex coloring problem, issued from classical graph theory, consists in choosing different colors for any two neighboring nodes in a arbitrary graph. In Distributed Computing, vertex coloring

[*]An extended abstract of this paper appeared in [GT00]

algorithms are mainly used for resource allocation (see [Lyn96] for more details). A vertex coloring defines a partial order on processors allowing them, for example, to execute their critical section according to the order defined by their respective colors.

Related problems include acyclic orientation of graphs (which can be induced by the partial ordering on vertices) and maximal independent set (which requires that no two neighboring vertices are colored black and that no extra vertex can be colored black without violating the first rule).

1.3 Related works

In uniform networks, it is well known that several problems cannot be solved self-stabilizingly using deterministic algorithms (*e.g.* [IJ90] shows that there exists no deterministic self-stabilizing mutual exclusion protocol for unidirectional uniform rings). Therefore, in the self-stabilizing setting, randomization was mostly used for symmetry breaking and construction of algorithms that self-stabilize with high probability (*e.g.* [BCD95, DGT00] both provide randomized self-stabilizing mutual exclusion protocols for unidirectional rings). Herman (in [Her92]) and Gradinariu and Tixeuil (in [GT00]) used randomization to reduce the memory space usually needed to solve the mutual exclusion problem and the l-mutual exclusion problem, respectively. Works by Israeli and Jalfon (see [IJ90]) and by Durand-Lose (see [DL98]) use randomization to weaken the scheduling requirements. A number of distributed algorithms are stabilizing only if the scheduling is constrained in scope (*e.g.* a single processor is allowed to perform an action at the same time) or in fairness (*e.g.* every processors performs an action infinitely often). Even with weaker scheduling requirements (where an arbitrary subset of the processors may perform an action at the same time, or where simple progression *vs.* fairness is needed), randomness sometimes permit that the solution remains self-stabilizing.

Self-stabilizing distributed vertex coloration was previously studied for planar and bipartite graphs (see [GK93, SS93, SRR94, SRR95]). Using a well-known result from graph theory, Gosh and Karaata [GK93] provide an elegant solution for coloring acyclic planar graphs with exactly six colors, along with an identifier based solution for acyclic orientation of planar graphs. This makes their solution limited to systems whose communication graph is planar and processors have unique identifiers. Sur and Srimani [SS93] vertex coloring algorithm is only valid for bipartite graphs. A paper by Shukla *et al.* (see [SRR95]) provides a randomized self-stabilizing solution to the two coloring problem for several classes of bipartite graphs, namely complete odd-degree bipartite graphs and tree graphs. Moreover, [SRR95] shows that there exist no deterministic self-stabilizing algorithm that provides a two coloring of an arbitrary odd-degree bipartite graph, even assuming the stronger scheduling hypothesis.

Recent works on self-stabilizing acyclic orientation have been presented in [DGPV00] for non-anonymous networks where a single vertex is distinguished. The maximal independent set problem was solved in [Lub86] using randomization yet was not self-stabilizing. In [SRR95], a self-stabilizing maximal independent set construction algorithm is given for general anonymous graphs, but assumes that processors are fairly scheduled.

1.4 Our contribution

We present three self-stabilizing solutions to the vertex coloring problem that perform in spite of unfair scheduling. The first two solutions are deterministic: one deals with anonymous networks but assumes a locally central unfair scheduler (that does not activates two neighboring processors at the same time), the other requires unique identifiers for processors but copes with unfair distributed scheduling. The last solution is randomized and presents weakest hypothesis: anonymous networks with unfair distributed scheduling.

Every solution do not need more than $B+1$ colors, where B denotes the network degree. Note that this bound is reached in the case of completely connected graphs. The time complexity is $O(n \times B)$ processor atomic actions. A nice property of our algorithm is that once stabilized, a directed acyclic orientation as well as a maximal independent set are obtained at no extra cost.

1.5 Outline

After defining the system setting in Section 2, we present our three solution to the coloration problem in Section 4, along with their proofs of correctness. Section 5 presents two direct applications for our work, while Section 6 provides concluding remarks.

2 Model

Distributed Systems We model a distributed system $\mathcal{S} = (C, T, I)$ as a *transition system* where C is the set of system configurations, T is a transition function from C to C, and I is the set of initial configurations. A *probabilistic distributed system* is a distributed system where a probabilistic distribution is defined on the transition function of the system.

We consider unidirectional ring networks where the processors maintain two types of variables: *local variables* and *field variables.* Each processor, P_i, has two neighbors named $left_i$ (its clockwise neighbor) and $right_i$ (its counter-clockwise neighbor). The local variables of P_i cannot be accessed by any of its neighbors, whereas the field variables of P_i are part of the shared register which is used to communicate with P_i's right neighbor. A processor can write only into its own shared register and can read only from the shared registers owned by its left neighbor or itself. The *state* of a processor is defined by the values of its local and field variables. A processor may change its state by executing its local *algorithm* (defined below). A *configuration* of a distributed system is an instance of the state of its processors.

The *algorithm* executed by each processor is described by a finite set of guarded actions of the form $\langle\text{guard}\rangle \longrightarrow \langle\text{statement}\rangle$. Each guard of processor P_i is a boolean expression involving P_i's variables and $left_i$'s field variables. A processor P_i is *enabled* in configuration c if at least one of the guards of the program of P_i is *true* in c. Let c be a configuration and CH be a subset of enabled processors in c. We denote by $\{c : CH\}$ the set of configurations that are *reachable* from c if every processor in CH executes an action starting from c. A *computation step* is a tuple (c, CH, c'), where $c' \in \{c : CH\}$. Note that all configurations $\in \{c : CH\}$ are reachable from c by executing *exactly one* computation step. In a probabilistic distributed system, every computation step is associated with a probabilistic value (the sum of the probabilities of the computation steps determined by $\{c : CH\}$ is 1). A *computation* of a distributed system is a maximal sequence of computation steps. A *history* of a computation is a finite prefix of the computation. A history of length n (denoted by h_n) can be defined recursively as follows:

$$h_n \equiv \begin{cases} (c_0, CH_0, c_1) & \text{if } n = 1 \\ [h_{n-1}(c_{n-1}, CH_{n-1}, c_n)] & \text{otherwise} \end{cases}$$

The probabilistic value of a history is the product of the probabilities of all the computation steps in the history. If h_n is a history such that

$$h_n = [(c_0, CH_0, c_1) \dots (c_{n-1}, CH_{n-1}, c_n)]$$

then we use the following notations: the length of the history h_n (equal to n) is denoted as $length(h_n)$, the last configuration in h_n (which is c_n) is represented by $last(h_n)$, and the first configuration in h_n (which is c_0) is referred to as $first(h_n)$ ($first$ can also be used for an infinite computation). A *computation fragment* is a finite sequence of computation steps. Let h be a history, x be a computation fragment such that $first(x) = last(h)$, and e be a computation such that $first(e) = last(h)$. Then $[hx]$ denotes a history corresponding to the computation steps in h and x, and (he) denotes a computation containing the steps in h and e.

3 Probabilistic Systems

In this section, we give an outline of the probabilistic model used in the rest of the paper. A detailed description of this model is available in [BGJ01].

Scheduler. A *scheduler* is a *predicate* over the system computations. In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a nonempty subset of the enabled processors in configuration c_i. In every computation step, this subset is chosen by the scheduler. The interaction between a scheduler and the distributed system generates some special structures, called *strategies*. The scheduler strategy definition is based on the tree of computations (all the computations having the same initial configuration). Let c be a system configuration and S a distributed system. The tree representing all computations in S starting from the configuration c is the tree rooted at c and is denoted as $\mathcal{T}ree(S, c)$. Let n_1 be a processor in $\mathcal{T}ree(S, c)$. A *branch* originating from n_1 represents the set of all $\mathcal{T}ree(S, c)$ computations starting in n_1 with the same first transition. The degree of n_1 is the number of branches rooted in n_1.

Definition 1 (Strategy) *Let S be a distributed system, D a scheduler, and c a configuration in S. We define a strategy as the set of computations represented by the tree obtained by pruning $Tree(S, c)$ such that the degree of any processor is at most* 1.

Definition 2 (Cone) *Let s be a strategy of a scheduler D. A* cone *$\mathcal{C}_h(s)$ corresponding to a history h is defined as the set of all possible computations under D which create the same history h.*

The probabilistic value of a cone $\mathcal{C}_h(s)$ is the probabilistic value of the history h (*i.e.*, the product of the probabilities of all computation steps in h).

Definition 3 (Subcone) *A cone $\mathcal{C}_{h'}(s)$ is called a* subcone *of $\mathcal{C}_h(s)$ if and only if $h' = [hx]$, where x is a computation fragment.*

Let S be a system, D a scheduler, and s a strategy of D. The set of computations under D that reach a configuration c' satisfying predicate P (denoted as $c' \vdash P$) is denoted as $\mathcal{EP}_s$, and its associated probabilistic value as represented by $Pr(\mathcal{EP}_s)$. We call a predicate P a *closed predicate* if the following is true: If P holds in configuration c, then P also holds in any configuration reachable from c.

Probabilistic Self-Stabilizing Systems. A probabilistic self-stabilizing system is a probabilistic distributed system satisfying two important properties: *probabilistic convergence* (the probability of the system to converge to a configuration satisfying a *legitimacy predicate* is 1) and *correctness* (once the system is in a configuration satisfying a legitimacy predicate, it satisfies the system specification). In this context, the correctness comes in two variants: *weak correctness*—the system correctness is only probabilistic, and *strong correctness*—the system correctness is certain.

Definition 4 (Strong Probabilistic Stabilization) *A system S is* strong self-stabilizing *under scheduler D for a specification SP if and only if there exists a closed legitimacy predicate L such that in any strategy s of S under D, the two following conditions hold:*
(i) The probability of the set of computations under D, starting from c, reaching a configuration c', such that c' satisfies L is 1 (probabilistic convergence). (Formally, $\forall s, Pr(\mathcal{EL}_s) = 1$).
(ii) All computations, starting from a configuration c' such that c' satisfies L, satisfy SP (strong correctness).(Formally, $\forall s, \forall e, e' \in s, e = (he') ::\ last(h) \vdash L \Rightarrow e' \vdash SP$).

Convergence of Probabilistic Stabilizing Systems. We borrow a result of [BGJ01] to prove the probabilistic convergence of the algorithms presented in this paper. This result is built upon some previous work on probabilistic automata ([PSL00, Seg95, SL94, WSS94]) and provides a complete framework for the verification of self-stabilizing probabilistic algorithms. We need to introduce a few terms before we are ready to present this result. First, we explain a key property, called *local convergence* and denoted by LC. Informally, the LC property characterizes a probabilistic self-stabilizing system in the following way: The system reaches a configuration which satisfies a particular predicate, in a bounded number of computation steps with a positive probability.

Definition 5 (Local Convergence) *Let s be a strategy, and P_1 and P_2 be two predicates on configurations, where P_1 is a closed predicate. Let δ be a positive number $\in]0,1[$ and N a positive integer. Let $\mathcal{C}_h(s)$ be a cone with $last(h) \vdash P_1$ and let M denote the set of subcones $\mathcal{C}_{h'}(s)$ of $\mathcal{C}_h(s)$ such that $last(h') \vdash P_2$ and $length(h') - length(h) \leq N$. Then $\mathcal{C}_h(s)$ satisfies the local convergence property denoted as $LC(P_1, P_2, \delta, N)$ if and only if $Pr(\bigcup_{\mathcal{C}_{h'}(s) \in M} \mathcal{C}_{h'}(s)) \geq \delta$.*

Now, if in strategy s, there exist $\delta_s > 0$ and $N_s \geq 1$ such that any cone $\mathcal{C}_h(s)$ with $last(h) \vdash P_1$ satisfies $LC(P_1, P_2, \delta_s, N_s)$, then the result of [BGJ01] states that the probability of the set of computations under D reaching configurations satisfying $P_1 \wedge P_2$ is 1. Formally:

Theorem 1 ([BGJ01]) *Let s be a strategy. Let P_1 and P_2 be closed predicates on configurations such that $Pr(\mathcal{EP}1_s) = 1$. If $\exists \delta_s > 0$ and $\exists N_s \geq 1$ such that any cone $\mathcal{C}_h(s)$ with $last(h) \vdash P_1$ satisfies $LC(P_1, P_2, \delta_s, N_s)$, then $Pr(\mathcal{EP}12_s) = 1$, where $P_{12} = P_1 \wedge P_2$.*

3.1 Distributed Systems

A distributed system is a set of state machines called processors. Each processor can communicate with a subset of the processors called neighbors. We will use $\mathcal{N}_x$ to denote the set of neighbors of node x. The communication among neighboring processors is carried out using the communication registers (called "shared variables" throughout this paper). The system's communication graph is drawn by representing processors as nodes and the neighborhood relationship by edges between the nodes.

Any processor in a distributed system executes an algorithm which contains a finite set of guarded actions of the form: $\langle label \rangle ::\ \langle guard \rangle \longrightarrow \langle statement \rangle$, where each guard is a boolean expression over the shared variables.

A *configuration* of a distributed system is an instance of the state of the system processors. A processor is *enabled* in a given configuration if at least one of the guards of its algorithm is *true*.

A distributed system can be modeled by a transition system. A transition system is a three-tuple $S = (\mathcal{C}, \mathcal{T}, \mathcal{I})$ where $\mathcal{C}$ is the collection of all the configurations, $\mathcal{I}$ is a subset of $\mathcal{C}$ called the set of initial configurations, and $\mathcal{T}$ is a function $\mathcal{T} : \mathcal{C} \longrightarrow \mathcal{C}$. A transition, also called a *computation step*, is a tuple (c_1, c_2) such that $c_2 = \mathcal{T}(c_1)$. A *computation* of an algorithm $\mathcal{P}$ is a *maximal* sequence of computations steps $e = ((c_0, c_1)\ (c_1, c_2)\ \ldots (c_i, c_{i+1}) \ldots)$ such that for $i \geq 0, c_{i+1} = \mathcal{T}(c_i)$ (a single *computation step*) if c_{i+1} exists, or c_i is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no processor is enabled in the terminal (final) configuration. All computations considered in this paper are assumed to be maximal.

A *history* of a computation is a finite prefix of a computation. A *fragment* of a computation e is a finite sequence of successive computation steps of e.

3.2 Scheduler

In this model, a *scheduler* is a *predicate* over the system computations. In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a nonempty subset of the enabled processors in configuration c_i. In every computation step, this subset is chosen by the scheduler. We refer to the following types of schedulers in this paper: *locally central scheduler* ([GH99, AN99, BDGM00]) in every computation step, neighboring processors are not chosen concurrently by the scheduler; *distributed scheduler* — during a computation step, any nonempty subset of the enabled processors is chosen by the scheduler.

The interaction between a scheduler and the distributed system generates some special structures called by us strategies. The strategy definition is based on the tree of computations. Let c be a system configuration. A *TS-tree* rooted in c, $\mathcal{T}ree(c)$, is the tree-representation of all computations beginning in c. Let n_1 be a node in $\mathcal{T}ree(c)$, a *branch* rooted in n_1 is the set of all $\mathcal{T}ree(c)$ computations starting in n_1 having the same first transition. The degree of n_1 is the number of branches rooted in n_1. A *sub-TS-tree of degree* 1 rooted in c is a restriction of $\mathcal{T}ree(c)$ such that the degree of any $\mathcal{T}ree(c)$'s node is at most 1. A strategy is defined as follows:

Definition 6 (Strategy) *Let TS be a transition system, let D be a scheduler and let c be a TS configuration. We call a scheduler strategy rooted in c a* sub-TS-tree *of degree 1 of $\mathcal{T}ree(c)$ such that any computation of the sub-tree verifies the scheduler D.*

Let st be a strategy. An *st-cone* $\mathcal{C}_h$ corresponding to a prefix h is the set of all possible st-computations with the same prefix h (for more details see [Seg95]). In the deterministic systems a cone of computations is reduced to a computation. The measure of an st-cone $\mathcal{C}_h$ is the measure of the prefix h (i.e., the product of the probability of every transition occurring in h). An st-cone $\mathcal{C}_{h'}$ is called a *sub-cone* of $\mathcal{C}_h$ if and only if $h' = [hx]$, where x is a computation factor.

3.3 Deterministic self-stabilization

In order to define self-stabilization for a distributed system, we use two types of predicates: the legitimacy predicate—defined on the system configurations and denoted by $\mathcal{L}$—and the problem specification—defined on the system computations and denoted by $\mathcal{SP}$.

Let $\mathcal{P}$ be an algorithm. The set of all computations of the algorithm $\mathcal{P}$ is denoted by $\mathcal{E}_{\mathcal{P}}$. Let $\mathcal{X}$ be a set and $Pred$ be a predicate defined on the set $\mathcal{X}$. The notation $x \vdash Pred$ means that the element x of $\mathcal{X}$ satisfies the predicate $Pred$ defined on the set $\mathcal{X}$.

Definition 7 (Deterministic self-stabilization) *An algorithm $\mathcal{P}$ is self-stabilizing for a specification $\mathcal{SP}$ if and only if the following two properties hold:*

1. *convergence — all computations reach a configuration that satisfies the legitimacy predicate. Formally,*
   $\forall e \in \mathcal{E}_{\mathcal{P}} :: e = ((c_0, c_1)(c_1, c_2) \ldots) : \exists n \geq 1, c_n \vdash \mathcal{C};$

2. *correctness — all computations starting in configurations satisfying the legitimacy predicate satisfy the problem specification $\mathcal{SP}$. Formally, $\forall e \in \mathcal{E}_{\mathcal{P}} :: e = ((c_0, c_1)\ (c_1, c_2)\ \ldots) : c_0 \vdash \mathcal{L} \Rightarrow e \vdash \mathcal{SP}$.*

3.4 Probabilistic self-stabilization

A predicate P is closed for the computations of a distributed system if and only if when P holds in a configuration c, P also holds in any configuration reachable from c.

Notation 1 *Let $\mathcal{S}$ be a system, D be a scheduler and st be a strategy satisfying the predicate D. Let CP be the set of all system configurations satisfying a closed predicate P (formally $\forall c \in CP, c \vdash P$). The set of st-computations that reach configurations $c \in CP$ is denoted by $\mathcal{EP}_{st}$ and its probability by $Pr_{st}(\mathcal{EP}_{st})$.*

In this paper we study silent algorithms - those for which the terminal configurations are legitimate. The probabilistic stabilization for this particular case of algorithms is restricted to the probabilistic convergence definition.

Definition 8 (Probabilistic Stabilization) *A system $\mathcal{S}$ is* self-stabilizing *under a scheduler D for a specification SP if and only if there exists a closed legitimacy predicate L on configurations such that in any strategy st of $\mathcal{S}$ under D, the two following conditions hold:*
The probability of the set of st-computations, starting from c, reaching a configuration c', such that c' satisfies L is 1 (probabilistic convergence). Formally, $\forall st, Pr_{st}(\mathcal{EL}_{st}) = 1$

3.5 Convergence of Probabilistic Stabilizing Systems

Building on previous works on probabilistic automata (see [SL94, WSS94, Seg95]), [BGJ99] presented a framework for proving self-stabilization of probabilistic distributed systems. In the following we recall a key property of the system called *local convergence* and denoted by LC.

Definition 9 (Local Convergence) *Let st be a strategy, $PR1$ and $PR2$ be two predicates on configurations, where $PR1$ is a closed predicate. Let δ be a positive probability and N a positive integer. Let $\mathcal{C}_h$ be a st-cone with $last(h) \vdash PR1$ and let M denote the set of sub-cones $\mathcal{C}_{h'}$ of the cone $\mathcal{C}_h$ such that the following is true for every sub-cone $\mathcal{C}_{h'}$: $last(h') \vdash PR2$ and $length(h') - length(h) \leq N$. The cone $\mathcal{C}_h$ satisfies LC $(PR1, PR2, \delta, N)$ if and only if $Pr(\bigcup_{\mathcal{C}_{h'} \in M} \mathcal{C}_{h'}) \geq \delta$.*

Now, if in strategy st, there exist $\delta_{st} > 0$ and $N_{st} \geq 1$ such that any st-cone, $\mathcal{C}_h$ with $last(h) \vdash PR1$, satisfies $LC(PR1, PR2, \delta_{st}, N_{st})$, then the main theorem of [BGJ99] states that the probability of the set of st-computations reaching configurations satisfying $PR1 \wedge PR2$ is 1.

4 Self-stabilizing coloration algorithms

In this section we provide two deterministic and one probabilistic solutions for the coloration problem. Our solutions are based on a greedy algorithm that takes the minimum available color.

The first deterministic algorithm (see Section 4.1) performs in anonymous networks yet requires a locally central scheduler. The second deterministic algorithm (see Section 4.2) makes use of unique identifiers but runs correctly under the unfair distributed scheduler. The probabilistic solution (see Section 4.3) offers the best of both worlds: unfair distributed scheduler support in anonymous networks.

4.1 Anonymous networks with locally central scheduler

The algorithm presented in this section requires a locally central scheduler, *i.e.* two neighboring nodes may not execute their critical section simultaneously. There exist numerous papers in the literature that provide such schedulers, *e.g.* [GH99], [AN99] and [BDGM00]. Our algorithm can be combined with any of those generic approaches to obtain a system that support stronger schedulers. Sections 4.2 and 4.3 provide alternative ways to obtain the same result.

4.1.1 Algorithm overview

Each processor maintains a *color*, whose domain is the set $\{0,\ldots,\delta\}$, where δ is the node's degree. The neighborhood agreement of a particular processor p is defined as follows:

Definition 10 (Agreement) *A processor p* agrees *with its neighborhood if the two following conditions are verified:*

1. *p's color is different from any of p's neighbors,*
2. *p's color is minimal within the set $\{0,...,\delta\} \setminus \cup_{j\in N_i}(R_j)$.*

When any of these two conditions is not verified, p performs the following actions: *(i)* p removes colors used by its neighbors from the set $\{0,\ldots,\delta\}$ and *(ii)* takes the minimum color of the resulting set as its new color. The resulting set is always non-empty. Core of the algorithm is presented in Algorithm 4.1.

For example, assume that the color set is $\{0,1,2,3\}$, p's color is 2 and p's neighbors use the colors 0 and 3. Then p does not agree with its neighborhood since Condition 2 of Definition 10 is not verified. After executing its algorithm, p's color becomes 1, the smallest element of the set $\{0,1,2,3\} \setminus \{0,3\} = \{1,2\}$.

Algorithm 4.1 Self-stabilizing Deterministic Coloration Algorithm

```
Shared Variable:
  $R_i$: integer $\in \{0,\ldots,\delta\}$;
Function:
  $Agree(i) : R_i = min\left(\{0,\ldots,\delta\} \setminus \bigcup_{j\in\mathcal{N}_i}\{R_j\}\right)$
Actions:
  $\mathcal{C} : \neg Agree(i) \longrightarrow R_i := min\left(\{0,\ldots,\delta\} \setminus \bigcup_{j\in\mathcal{N}_i}\{R_j\}\right)$
```

4.1.2 Algorithm analysis

In this section, we first define legitimate configurations as configurations where every processor agrees with its neighborhood. Since any terminal configuration of Algorithm 4.1 is legitimate, we concentrate on proving convergence from any initial configuration.

Definition 11 *A configuration is* legitimate *if and only if every processor p agrees (in the sense of Definition 10) with its neighborhood.*

In an arbitrary initial configuration c, a processor p may not agree with its neighborhood. From Definition 10, this may occur in two (non mutually exclusive) cases:

1. **First kind disagreement:** there exists some $q \in \mathcal{N}_p$ such that $R_p = R_q$. Let M_1^c be the set of such processors p in c.
2. **Second kind disagreement:** there exists a color C in $\{0, \ldots, \delta_p\} \setminus \bigcup_{q \in \mathcal{N}_p} \{R_q\}$, such that $R_p < C$. Let M_2^c be the set of such processors p in c.

We first show that for any processor p in M_1^c, executing its action leads to a configuration c' where $p \notin M_1^{c'}$ (see Lemma 1). Then we show that for any processor p in $M_2^{c'}$ yet not in $M_1^{c'}$, the number of executed actions in any computation is bounded (see Lemma 2). We conclude that overall any system computation ends up in a terminal configuration, which is legitimate (see Definition 11).

Lemma 1 *Let $e = ((c_1, c_2), \ldots, (c_k, c_{k+1}), \ldots)$ be a computation of Algorithm 4.1 under a locally central scheduler. If (c_k, c_{k+1}) is an action of a processor of $M_1^{c_k}$, then for any $i > k$, $|M_1^{c_k}| > |M_1^{c_i}|$.*

Proof: Let $p \in M_1^{c_k}$ be a processor which executes Rule $\mathcal{C}$ at c_k. None of p's neighbors may execute an action (by the locally central scheduler hypothesis), and Rule $\mathcal{C}$ gives p a color that is different from any of its neighbors in c_{k+1}, therefore $|M_1^{c_k}| > |M_1^{c_{k+1}}|$.

For any processor p, it is impossible that Rule $\mathcal{C}$ results in giving the same color to p that any of its neighbors, thus for any $i \geq k+1$, $|M_1^{c_{k+1}}| \geq |M_1^{c_i}|$. □

A direct consequence of this proof is that any processor executes its action at most once for being in M_1.

Lemma 2 *Let $e = ((c_1, c_2), \ldots, (c_k, c_{k+1}), \ldots)$ be a computation of Algorithm 4.1 under a locally central scheduler. If (c_k, c_{k+1}) is an action of a processor p of $M_2^{c_k}$ not in $M_1^{c_k}$, then p may only execute $B-2$ actions in any subsequent computation.*

Proof: Let $p \in M_2^{c_k} \setminus M_1^{c_k}$ be a processor which executes Rule $\mathcal{C}$ at c_k. None of p's neighbors may execute an action (by the locally central scheduler hypothesis), and Rule $\mathcal{C}$ gives p a color that is strictly smaller than its previous one (its previous color was not minimal). Since its previous color was at most δ, its new color is at most $\delta - 1$. Since p's color may only decrease to reach 0 and that Rule $\mathcal{C}$ strictly increases p's color, then starting from c_{k+1}, p may only execute its action at most $B-2$ times. □

Theorem 2 *Any computation of Algorithm 4.1 under a locally central scheduler eventually achieves a legitimate configuration.*

Proof: Let e be a computation of Algorithm 4.1 under a locally central scheduler starting in the configuration c. By Lemmas 1 and 2, a processor p may execute at most $B-1$ actions. Then after at most $n \times (B-1)$ actions, the system reaches a terminal configuration, where no rule is enabled. Since any terminal configuration is legitimate, the theorem is proved. □

4.2 Identifier networks with unfair distributed scheduler

In this section, we transform Algorithm 4.1 such that it stabilizes in spite of any unfair distributed scheduler. In order to break possible network symmetry we make use of unique processor identifiers. In actual networks, such identifiers can be obtained from the network device.

4.2.1 Algorithm overview and analysis

Algorithm 4.2 differs from Algorithm 4.1 in two ways:

1. processors that are colored with the same color as one of their neighbors may execute Rule $\mathcal{C}_1$ if and only if their identifier is locally minimal between all identically colored neighbors,
2. processors colored with a color different from any of their neighbors may execute rule $\mathcal{C}_2$ as in Algorithm 4.1.

Algorithm 4.2 Self-stabilizing Deterministic Coloration Algorithm under an unfair scheduler

```
Shared Variable:
  $R_i$: integer $\in \{0, \ldots, B\}$;
Function:
  $Agree(i) : R_i = min\left(\{0, \ldots, B\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
Actions:
  $\mathcal{C}_1 : \neg Agree(i) \wedge$
    $(\exists j \in \mathcal{N}_i, R_j = R_i \wedge id_i > min\,(id_k, k \in \mathcal{N}_i \wedge R_i = R_k)) \longrightarrow$
        $R_i := min\left(\{0, \ldots, B\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
  $\mathcal{C}_2 : \neg Agree(i) \wedge (\forall j \in \mathcal{N}_i,\ R_i \neq R_j) \longrightarrow$
        $R_i := min\left(\{0, \ldots, B\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
```

We use the same proof technique as that of Algorithm 4.2 by showing that any processor is able to perform a bounded number of actions, implying that any computation of the system is finite.

For technical reasons, we split the set of processors in three mutually exclusive sets:

- S_1 — the set of processors having the same color as one of their neighbors. Formally,

$$S_1 = \{i \mid \neg Agree(i) \wedge \exists j \in \mathcal{N}_i,\ R_i = R_j\}$$

- S_2^k — the set of the k-colored processors ($0 \leq k < B$) that do not agree with their neighbors and whose color is different from those of their neighbors. Formally,

$$S_2^k = \{i \mid \neg Agree(i) \wedge R_i = k \wedge (\forall j \in \mathcal{N}_i,\ R_i \neq R_j)\}$$

- S_3^k — the set of the k-colored processors ($0 \leq k < B$) that agree with their neighbors. Formally,

$$S_3^k = \{i \mid Agree(i) \wedge R_i = k\}$$

The first two sets we consider processors with some kind of disagreement (see Section 4.1.2), while the third set includes processors that agree with their neighbors. We use these sets when proving that any system computation eventually leads to a configuration where *all* processors are in the S_3^j sets ($0 \leq j \leq B$). In such a configuration, no rule can be executed and the configuration is terminal.

In more details, we first show that a processor of S_1 may execute Rule $\mathcal{C}_1$ and eventually become an element of S_3^k. Then, a processor of S_2^k may execute Rule $\mathcal{C}_2$ and then become a member of S_1 or S_3^j (with $j > k$). In turn, a processor of S_3^k may either remain forever in this set of move to set S_2^k if one of its neighbors, by executing Rule $\mathcal{C}_2$, frees a color greater than k. Since the number of sets S_3^k and S_2^k ($0 \leq k \leq B$) is finite then, eventually, a terminal configuration is reached.

Lemma 3 *Let e be a computation of Algorithm 4.2 starting in a configuration where $|S_1|+\sum_{k=0}^{B-1}|S_2^k| \neq 0$. Then e eventually reaches a configuration where $|S_1|+\sum_{k=0}^{B-1}|S_2^k| = 0$.*

Proof: Let c be the initial configuration of e. We study the value of $|S_1|+\sum_{k=0}^{B-1}|S_2^k|$ after execution of some processor p action in c:

1. $p \in S_1$, and $\forall q \in \mathcal{N}_p, q \in S_2^t, (0 \leq t \leq B-1)$. Then p executes Rule $\mathcal{C}_1$ and moves to S_3^k;
2. $p \in S_1$, and $\exists q \in S_2^r$ such that p and q are chosen by the scheduler at the same time and simultaneously execute their action. After execution of Rule $\mathcal{C}_1$, p has two possibilities: *(i)* stepping out of S_1 or *(ii)* coloring itself with a greater color s.
3. $p \in S_2^r$ and $\forall q \in \mathcal{N}_p, q \in S_1$ or $q \in S_2^m (m \leq r)$, q does not execute its action at c. Then, p may only execute Rule $\mathcal{C}_2$ and move to S_3^t, with $t > r$
4. $p \in S_2^r$ and $\exists q \in \mathcal{N}_p, q \in S_1$ or $q \in S_2^m (m \leq r)$ such that p and q are chosen by the scheduler at the same time and simultaneously execute their action. After executing Rule $\mathcal{C}_2$, p can either move to S_3 or to S_1, colored with $s > r$ as its neighbor q. Then, there are two possible cases:
   (a) After execution of Rule $\mathcal{C}_1$, p's neighbors may choose a color that is different from p, which makes p an element of S_3^s.
   (b) If p has the minimal identifier between its s-colored neighbors colored, then only p may execute an action in its neighborhood and move to S_3^k.

Note that a processor may move from S_3^k ($0 \leq k \leq B-1$) to S_2^k only if one of its neighbors (in S_2^t, with $t > k$) executes Rule $\mathcal{C}_2$ so that color t becomes available to p.

Now assume that there exists a processor q that executes actions infinitely. We consider an execution starting in configuration c' where $q \in S_1$ is s-colored and chosen to execute its action. Then q colors itself with k_1. Processor q would move again to S_1 if there exists a free color greater than k_1. By hypothesis, q executes infinitely many actions. Using a similar argument as in the proof of Lemma 2, q may move to S_1 only a finite number of times, hence our hypothesis is false, and the expression $|S_1|+\sum_{k=0}^{B-1}|S_2^k|$ eventually decreases. Since the system only terminates when all processors are in some S_3, the preceding sum eventually reaches 0. □

4.3 Anonymous networks with unfair distributed scheduler

In this section we present the randomized variant of Algorithm 4.1. This algorithm works on anonymous networks and stabilizes with an unfair scheduler.

Algorithm 4.3 Self-stabilizing Randomized Coloration Algorithm

```
Shared Variable:
  $\forall j \in \mathcal{N}_i, R_j$: integer $\in \{0, \ldots, B\}$;
Function:
  $Agree(i) : R_i = min\left(\{0, \ldots, B\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
Actions:
  $\mathcal{C} : \neg Agree(i) \longrightarrow$
            if random(0,1)=1 then
                $R_i := min\left(\{0, \ldots, B\} \setminus \bigcup_{j \in \mathcal{N}_i} \{R_j\}\right)$
```

4.3.1 Algorithm overview and analysis

Compared to Algorithm 4.1, a processor which does not agree with one of its neighbors tosses a coin before changing its color. Even if neighboring processors would compete for executing their action, by randomization there exists a positive probability that only one of those processors executes its actions.

In order to prove the correctness of Algorithm 4.3, we study an arbitrary strategy of this algorithm under the distributed unfair scheduler. We prove that in this strategy, the set of computations achieving a terminal configuration in a finite number of computation steps has a positive probability. Hence the strategy satisfies the local convergence property (see Definition 9) and the set of computations reaching terminal configuration has probability 1. The proof is divided in two main parts:

1. starting in an arbitrary configuration, the system eventually reaches a configuration where all processors have a color that is different from their neighbors;

2. starting in such a configuration, the system eventually reaches a configuration where all processors agree with their neighbors (see Definition 10).

Lemma 4 *In any strategy st of Algorithm 4.3 under the unfair distributed scheduler, there exists a positive probability to achieve a legitimate configuration in a finite number of steps.*

Proof: Let c be a starting configuration for the strategy. Assume that in c, both M_1^c and M_2^c (see Section 4.1.2 for definition) are non-empty. We now prove the two previously outlined parts.

We consider the following scenario for the first part: *(i)* every time when some neighboring processors are chosen simultaneously by the scheduler to execute their action, exactly one of them execute its action, and *(ii)* only processors which neighbors have the same color execute their rule. Note that Condition *(i)* of this scenario simulates the locally central scheduler.

This scenario repeats itself until there are no two neighboring processors colored identically. Let us denote by c' a configuration where $|M_1^{c'}| = 0$. In Strategy st, the probability of the set of computations reaching c' is

$$\epsilon_1 \geq \left(\frac{1}{2}\right)^n \times \left(\frac{1}{2}\right)^{\sum_{i=1}^n d_i}$$

where n is the network size and d_i is the degree of the node i. The lower bound for the probability value is obtained by considering that a processor i executes its rule and none of its neighbors execute their rule with probability $\frac{1}{2} \times \left(\frac{1}{2}\right)^{d_i}$, and that there are at most n processors in the network.

The scenario for the second part is reduced to Condition *(i)* of the first scenario. According to Lemma 2, a processor can only execute a finite number of actions (bounded by $B-2$). Therefore the the set of computations reaching c'' (with $|M_1^{c''}| = 0$ and $|M_2^{c''}| = 0$) has probability

$$\begin{array}{rcl} \epsilon & \geq & \epsilon_1 \times \left(\frac{1}{2}\right)^{(B-2)\times n} \times \left(\frac{1}{2}\right)^{(B-2)\times \sum_{i=1}^{n} d_i} \\ & \geq & \left(\frac{1}{2}\right)^{(B-1)\times(n+\sum_{i=1}^{n} d_i)} \end{array}$$

where n is the network size and d_i is the degree of the node i. □

Lemma 5 *The average number of computations steps to reach a configuration c where all processors agree with their neighbors is $O((B-1)\times \log_2 n)$.*

Proof: Let A be the set of processors which agree with their neighbors (see Definition 10). By Lemmas 1 and 2, the probability for processor i moving to A after at most $B-1$ trials is

$$p_i \geq \left(\frac{1}{2}\right)^{B-1} \times \left(\frac{1}{2}\right)^{B\times(B-1)}$$

Therefore, for n-sized networks, the average number of processors in A after $B-1$ trials is at least $n \times \left(\frac{1}{2}\right)^{(B+1)\times(B-1)}$. This also means that at most $n \times \left(1 - \left(\frac{1}{2}\right)^{(B^2-1)}\right)$ processors are not in A.

After $x \times (B-1)$ trials, the average number of processors in A is at least $n \times \left(1 - \left(\frac{1}{2}\right)^{(B^2-1)}\right)^x$. The algorithm would stop when all processors agree. Then x is a solution of the following equation

$$\begin{array}{rcl} \left[n \times \left(1 - \left(\frac{1}{2}\right)^{(B^2-1)}\right)^x = 1\right] & \Rightarrow & \left[x = \log_{\frac{1}{1-\frac{1}{2}^{(B^2-1)}}} n\right] \\ & \Rightarrow & \left[x = \frac{\log_2 n}{\log_2 \frac{1}{1-\frac{1}{2}^{(B^2-1)}}}\right] \\ & \Rightarrow & x = O(\log_2 n) \end{array}$$

Therefore, on average, all processors agree with their neighbors within $O((B-1)\times \log n)$ computation steps. □

5 Applications

In this section we present two immediate applications of our algorithms: acyclic orientation and maximum independent set. In the following, we assume that each processor i has a color R_i that satisfies $Agree(i)$ (see Definition 10). Depending on the scheduling and system symmetry, one of our algorithms will be used. In the following, we refer those algorithms under the common name of *Coloring Algorithm*.

5.1 Acyclic orientation

A directed acyclic graph (or DAG) can be derived from any terminal configuration of our coloring algorithm by using the following predicate:

Definition 12 *Let c be a terminal configuration of the coloring algorithm. Let (i,j) be an edge of the communication graph. The edge (i,j) is* oriented *from i to j if in c, $R_i < R_j$.*

That definition was used in [GK93, DDT99] with system-wide unique identifier. The following lemma states that local coloration is sufficient.

Lemma 6 *In any terminal configuration of the coloring algorithm, Definition 12 induces an acyclic orientation.*

Proof: Let c be a terminal configuration of the coloring algorithm. Suppose that Definition 12 induces a cycle in the communication graph in c. Let $p_1,\ldots,p_m$ the processors in this cycle. By Definition 12, we would then have $R_1 < R_1$, which is impossible. $\Box$

All previously known self-stabilizing algorithms that require directed acyclic graphs (such as those presented in [DDT99]) can be run on top of the coloring algorithm to obtain the same results on anonymous networks.

5.2 Maximal independent set

Solving the maximal independent set problem enables to construct a set $\mathcal{M}$ of processors such that the following two conditions are satisfied:

1. no two neighboring processors are in $\mathcal{M}$,
2. there is no other set $\mathcal{M}'$ such that $\mathcal{M} \subset \mathcal{M}'$ and no two neighboring processors are in $\mathcal{M}'$.

In this section we prove that a maximal independent set can be derived from any terminal configuration of our coloring algorithm by using the following predicate:

Definition 13 *Let c be a terminal configuration of the coloring algorithm. Let $\mathcal{M}_c$ be the set of processors colored with B (where B is the bound used by the coloring algorithm).*

Lemma 7 *In any terminal configuration of the coloring algorithm, Definition 12 induces a maximal independent set.*

Proof: Assume that there exists another set $\mathcal{M}'_c$ of independent processors such that $\mathcal{M}_c \subset \mathcal{M}'_c$. This means that there exists at least one processor p in $\mathcal{M}'_c$ that is not in $\mathcal{M}_c$. Let us enumerate the different possibilities:

1. processor p is colored with B and then $\mathcal{M} = \mathcal{M}'$ or
2. processor p has no neighbor in $\mathcal{M}$ and it is not colored with B, which means that $Agree(p)$ is false that Configuration c is not terminal.

Any of those two case contradicts the hypothesis. $\Box$

Unlike the maximal independent set algorithm provided in [SRR95], we do not assume that the scheduler is fair between system processors. Only simple progression is needed to ensure system stabilization. The cost for this extra convenience is the additional memory space (that was $O(1)$ in [SRR95]).

6 Conclusions

We provided three self-stabilizing solutions to the vertex coloring problem that perform in spite of unfair scheduling. In particular, the last solution is randomized and presents weakest hypothesis: anonymous networks with unfair distributed scheduling. As direct application, we were able to solve directed acyclic orientation as well as maximal independent set at no additional cost.

References

[AN99] A. Arora and M. Nesterenko. Stabilization-preserving atomicity refinement. *DISC'99*, pages 254–268, 1999.

[BCD95] J. Beauquier, S. Cordier, and S. Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proceedings of the Second Workshop on Self-stabilizing Systems (WSS'95)*, pages 15.1–15.15, 1995.

[BDGM00] J. Beauquier, A. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *Proceedings of DISC'2000*, page to apear, October 2000.

[BGJ99] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing optimal leader election under arbitrary scheduler on rings. Technical Report 1225, LRI, September 1999.

[BGJ01] J Beauquier, M Gradinariu, and C Johnen. Crossover composition. In *Proceedings of the Fifth Workshop on Self-stabilizing Systems (WSS 2001)*, pages 19–34, 2001.

[DDT99] S. K. Das, A. K. Datta, and S. Tixeuil. Self-stabilizing algorithms on dag structured networks. *Parallel Processing Letters*, 9(4):563–574, December 1999.

[DGPV00] A. K. Datta, S. Gurumurthy, F. Petit, and V. Villain. Self-stabilizing network orientation algorithms in arbitrary networks. In *Proceedings of the Twenteeth International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 576–583, 2000.

[DGT00] A. K. Datta, M. Gradinariu, and S. Tixeuil. Self-stabilizing mutual exclusion using unfair distributed scheduler. In *Proceedings of the IPDPS'2000 International Conference*, pages 465–470, Cancun, Mexico, May 2000.

[Dij74] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[DL98] J. Durand-Lose. Randomized uniform self-stabilizing mutual exclusion. In *Proceedings of the Second International Conference on Principles of Distributed Systems (OPODIS'98)*, pages 89–98, 1998.

[GH99] M. Gouda and F. Hadix. The alternator. *In Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53, 1999.

[GK93] S. Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, pages 7:55–59, 1993.

[GT00] M. Gradinariu and S. Tixeuil. Tight space uniform self-stabilizing l-mutual exclusion. Technical Report 1249, LRI, March 2000.

[Her92] T. Herman. Self-stabilization: randomness to reduce space. *Distributed Computing*, 6:95–98, 1992.

[IJ90] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the International Conference on Principles of Distributed Computing (PODC'90)*, pages 119–131, 1990.

[Lub86] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *SIAM, Journal of Computing*, volume 15(4), pages 1036–1053, 1986.

[Lyn96] N Lynch. Distributed algorithms. *Morgan Kaufmann*, 1996.

[PSL00] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of Aspen and Herlihy: a case study. *Distributed Computing*, 13(4):155–186, 2000.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[Seg95] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, MIT, Departament of Electrical Engineering and Computer Science, 1995.

[SL94] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In Springer-Verlag, editor, *Proceedings of the Fifth International Conference on Concurrency Theory (CONCUR'94) LNCS:836*, Uppsala, Sweden, August 1994.

[SRR94] S. Shukla, D. Rosenkrantz, and S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proceedings of the International Workshop on Parallel Processing*, pages 668–673, Bangalore, India, 1994. Tata-McGrawhill, New Delhi.

[SRR95] S. Shukla, D. Rosenkrantz, and S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the Second Workshop on Self-stabilizing Systems (WSS'95)*, pages 7.1–7.15, 1995.

[SS93] S. Sur and P. K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences*, 69:219–227, 1993.

[WSS94] S. H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic i/o automata. In *Proceedings of the Fifth International Conference on Concurrency Theory (CONCUR'94) LNCS:836*, pages 513–528, 994.