Computing automatically the stabilization time against the worst and the best schedulers

Rapport de Recherche LRI n^o 1448

Joffroy Beauquier, Colette Johnen, Stéphane Messika

L.R.I./C.N.R.S., Université de Paris-Sud,
bat 490, 91405 Orsay Cedex, France
jb@lri.fr, colette@lri.fr, messika@lri.fr

abstract: In this paper, we reduce the problem of computing the convergence time for a randomized self-stabilizing algorithm to an instance of the stochastic shortest path problem (SSP). This problem has been solved, and the solution gives us a way to compute automatically the stabilization time against the worst and the best scheduler. Moreover, a corollary of this reduction ensures that the best and the worst schedulers for this kind of algorithms are memoryless and deterministic. We apply these results here in several examples.

key words: randomized algorithms, distributed algorithm, self-stabilizing system, scheduler.

Résumé: Nous réduisons le problème du calcul du temps de converge d'un algorithme probabiliste et auto-stabilisant à une instance du problème SSP - problème du plus court chemin stochastique. Le problème SSP admet une solution calculable automatiquement. Nous avons ainsi un moyen d'établir automatiquement le temps moyen de stabilisation dans le cas où l'ordonnenceur a le pire comportement, et aussi dans le cas où il a le meilleur comportement. De plus, un corollaire de cette réduction assure que l'ordonnencement conduisant au pire temps moyen de stabilisation et celui qui conduit au meilleur temps moyen sont déterministes et sans mémoire. Nous appliquons ses résultats sur plusieurs algorithmes probabiliste et auto-stabilisants.

Mots clés: algorithmes probabiliste, algorithme répartie, auto-stabilization, ordonnanceur.

1 Introduction

By their very nature, distributed algorithms have to deal with a non-deterministic environment. The speeds of the different processors or the message delays are generally not known in advance and may vary substantially from one execution to the other. For representing the environment in an abstract way, the notion of scheduler (also called demon or adversary) has been introduced. The scheduler is in particular responsible of which processors take a step in a given configuration or of which among the messages in transit arrives first. It is well known that the correctness of a distributed algorithm depends on the considered scheduler. This remark also holds for self-stabilizing distributed algorithm.

Different classes of schedulers have been considered in the literature. With the synchronous scheduler, all enabled processors take an elementary step concurrently, with the central scheduler (central demon) the processors take their steps one after the other, with a distributed scheduler, a subset of enabled processors take a step concurrently and with the probabilistic scheduler all enabled processors take a step with some given probability. If there is a unique synchronous scheduler, there is an infinity of distributed schedulers (corresponding to all possible choices of subsets along the computation). In this paper, we restrict our attention to probabilistic self-stabilization. Classically self-stabilization requires convergence (each execution reaches a legitimate configuration) and correctness (each execution starting from a legitimate configuration satisfies the specification). Probabilistic self-stabilization requires that convergence and correctness are only probabilistic (this notion will be presented in a formal way later). It appears that the convergence property of a given algorithm depends on the chosen scheduler. With some schedulers the algorithm can converge in a finite bounded number of steps (the stabilization time) while with others it can not converge at all. Even if the stabilization time is finite, it can differ according to the scheduler. There is generally infinity many schedulers in a given class, as for the class of distributed schedulers that we consider here. It is thus interesting to know a best scheduler (the scheduler that gives the smaller expected stabilization time) and a worst. Note that a best scheduler can possibly give a finite stabilization time and a worst an infinite one. In some cases best and worst both give finite stabilization time (it is then said that the algorithm is self-stabilizing under "the" distributed scheduler).

One could think that the best and the worst schedulers are intricate and difficult to describe, or that their simulation would use a lot of resources. In fact it is not true because there are always a best scheduler and a worst scheduler that are memoryless (meaning that the choice they make in a given configuration depends only on the configuration, and not on the past of the execution). This results extends a result in [3] and, although in a different context, a result of [2]. But we do much more: we give an automatic way of computing a) both of these schedulers, under the form of Markov processes, b) the expected stabilization time (when it is finite, elsewhere we get that the algorithm does not converge) for both of the schedulers.

We give three examples. The first one is very simple. Guessing the best and the worst scheduler is easy and (fortunately!) the automatic construction gives the same result. Guessing the schedulers on the two other examples is not so easy, and we hope that the reader will be convinced he couldnt get the schedulers for the third example "by hand". We also show in each case how the stabilization time can be computed using a fixpoint method.

Related works. In [4], Dolev, Israeli and Moran introduced the idea of a two players game between the scheduler and what they call luck, i.e. the random values, without defining formally the probabilistic space of computations. The structure (informally presented) behind a sl-game is a strategy (formally defined in this paper) where some branches have being cut. In [12], [10], and [11], Lynch, Pogosyants and Segala present a formal method for analyzing probabilistic I/O automata which modelize distributed systems. A clear distinction between the algorithm, which is probabilistic, and the scheduler, which is non-deterministic, is made. The notion of cone, that is at the basis of the probabilistic space, is also used. These works do not consider self-stabilization. In [6, 5] the notion of randomized distributed algorithms under a fixed scheduler are studied using methods issued from Markov theory, in [5] the authors present how to adapt these methods for an arbitrary scheduler. Moreover, de Alfaro in [3] applies analogies with SSP to compute maximal and minimal reachability probability for probabilistic transition systems.

2 Notion of Markov Decision Processes

In this section we adopt the notation of de Alfaro [3].

Informally, a Markov decision process is a generalization of the notion of Markov chain in which a set of possible actions is associated to each state. To each state-action pair corresponds a probability distribution on the states, which is used to select the successor state. A Markov chain corresponds thus to a Markov decision process in which there is exactly one action associated with each state. The formal definition is as follows.

Definition 1 (Markov Decision Process) *A Markov decision process (MDP) (S, Act, A, p) consists of a finite set S of states, a finite set Act of actions, and two components A, p that specify the transition structure.*

- *For each $s \in S$, $A(s)$ is the non-empty finite set of actions available at s.*
- *For each $s, t \in S$ and $a \in A(s)$, $p_{st}(a)$ is the probability of a transition from s to t when action a is selected. Moreover, p verifies the following property $\forall s, \forall a \in A(s)$ we have $\sum_{t \in S} p_{st}(a) = 1$.*

Definition 2 (Behavior of MDP) *A behavior of a Markov decision process is an infinite sequence of alternating states and actions, constructed by iterating a two phase selection process. First, given the current state s, an action $a \in A(s)$ is selected non deterministically; second the successor state t of s is chosen according to the probability distribution $P(t|s,a) = p_{st}(a)$.*
Given a state s we denote Ω_s the set of all the behaviors starting in s.

Definition 3 (cylinder sets) *The basic cylinder associated to the history $h = s_0 a_0 s_1 a_1 ... s_n$ contains all behaviors of a MDP starting at s_0 having the same prefix h:*

$$C_h = \{w \in \Omega_s | X_0 = s \wedge Y_0 = a_0 \wedge ... \wedge X_n = s_n \wedge Y_n = a_n\}$$

We have to define also some measurable sets of behaviors. For every state s, we let $B_s \in 2^{\Omega_s}$ be the smallest algebra of subsets of Ω_s, that contains all the basic cylinder sets and that is closed under complement and countable unions and intersections. This algebra is called the Borel *σ-algebra* of basic cylinder sets, and its element are the measurable sets of behaviors (see [3]).
To be able to talk about the probability of behaviors, we would like to associate to each $\Delta \in B_s$ a probability measure $P(\Delta)$. However this measure is not well defined, since the probability that a

behavior belongs to Δ depends on how the actions have been nondeterministically chosen.
To represent these choices, we use the concept of policy (see [3]). Policies are closely related to the adversaries of Segala and Lynch [12] and to the schedulers of Lehman and Rabin [8], Vardi [13] and Pnueli and Zucck [9].

Definition 4 (Policy) *A policy η is a set of conditional probabilities $Q_\eta(a|s_0 s_1 ... s_n)$, for all $n \geq 0$, all possible sequences of states $s_0, ..., s_n$ and all $a \in A(s_n)$. It must be $0 \leq Q_\eta(a|s_0, s_1..., s_n) \leq 1$ and $\sum_{a \in A(s_n)} Q_\eta(a|s_0, s_1..., s_n) = 1$.*
A policy is deterministic if for each s there is an action $a \in A(s)$ such that $Q_\eta(a|s_0, s_1..., s_n) = 1$.
A policy is called Markovian if the choice of the action at a state does not depend on the portion of behavior before the state is reached.

Some can object that this policy does not take care about the history of the actions but De Alfaro in [3] has already shown that all the definitions are equivalent. Informally, a policy dictates the probabilities with which the actions are chosen knowing all the history of the process.

Definition 5 (Probability measure of a cylinder under a policy) *Let η be a policy. Let $h = s_0 a_0 s_1 a_1 ... s_n$ be an history.*

$$P_s^\eta(C_h) = \prod_{k=0}^{n-1} p_{s_k s_{k+1}}(a_k) Q_\eta(a_k|s_0, s_1..., s_k)$$

There is an unique extension of the probabilistic measure P_s^η, to any element of B_s. Thus the triple (η, B_s, P_s^η) defines a probabilistic space on B_s.

3 Randomized Distributed Algorithms as Markov Decision Processes

We will present, here, how we can modelize a randomized distributed algorithm as a Markov Decision Process. One can refer to [6, 5] to get more details.

In a distributed system, the topology of the network of machines is generally given under the form of a graph $G = (V, E)$, where the set $V = \{1, ..., N\}$ of vertices corresponds to the location of the machines. There is an edge between two vertices when the corresponding machines can communicate together. All the machines are identical finite state machines. The space of states is Q. A configuration X of the distributed system is the N-tuple of all the states of the machines. The set of configurations Q^N is denoted ϵ. Given a configuration X of ϵ, the state of the i^{th} machine is written $X(i)$. Randomized distributed algorithms are characterized by a scheduler (adversary), i.e., a mechanism which selects, at each step, a nonempty subset of enabled machines which apply the guarded rules of the algorithm. The execution simultaneously by several machines of rules is call a *computation step*. Let us recall that for a given fixed memoryless (Markovian) scheduler, the randomized distributed algorithm can be seen as a Markov chain $X_t = (X_t(1), ...X_t(N))$ where X_t is a random variable taking its values in ϵ (see [6]).

Here, we consider that the scheduler is arbitrary, then given a configuration s we define $A(s)$ by all the non-empty subsets of machines enabled in this configuration. With this definition we see that the notion of scheduler corresponds to the notion of determistic policy. Then, the randomized distributed algorithm can be seen as a Markov Decision Process.

Example 1 (Toy Example) *Here we define a toy example that will help us to illustrate the results of the paper. This algorithm presented achieves token circulation on unidirectional ring (see,[7, 1]):*

$$Token_p \rightarrow \textit{if } (random(0,1) = 0) \textit{ then } Pass\_Token_p.$$

The complete algorithm 4 is presented in annexe.

The algorithm to be self-stabilizing should converge from a configuration with two tokens to a configuration with one token. This algorithm ensures that in any configuration, the ring has at least one token.

4 Finding best and worst Schedulers for Self-Stabilization Algorithms

In this paper, we will focus on randomized self-stabilizing algorithms. We can recall here the definition.

4.1 Probabilistic self-stabilization

Definition 6 *A randomized algorithm A is self-stabilizing towards a set of legitimate states L if*

- *Whatever the starting configuration s_0 it reaches a configuration in L within a finite number of states and with probability* 1
- *The set L of states is closed under the rules of A*

Example 2 *Under some schedulers (for instance a fixed one choosing one token after the other), it is very easy to check that our toy example converges towards the set of configurations with only one token.*

The aim of the following sections is to adapt some results on stochastic shortest path problem of de Alfaro [3] and Bertsekas and Tsitsiklis [2] to randomized self-stabilizing algorithms.

4.2 Results on the Stochastic Shortest Path Problem

We just present here the main results, but we refer to [3, 2, 6] to more details. Informally, the stochastic shortest path problem consists in computing the minimum expected cost for reaching a given subset of destination states, from any state of a Makov decision process (in which a cost is associated to each action). This problem was first study by Bertsekas and Tsitsiklis in [2] and by de Alfaro in [3]. The following notations and definition are taken from [3] and [2].

Definition 7 (Instance of Stochastic Shortest Path Problem) *An instance of the stochastic shortest path problem consists of an MDP $\Pi = (M, U, c, g)$ in which the components M U, c and g are defined as follows:*

- *M is Makovien Decision Process*
- *U is the set of destination states*
- *c is the cost function, which associates to each state $s \in S - U$ and action $a \in A(s)$ the cost $c(s, a)$*
- *g is the terminal cost function which associates to each $s \in U$ its terminal cost $g(s)$.*

Definition 8 (SSP-proper policy) *T_U is a random variable indicating the position of the first entrance in U: $T_U = min\{k | s_k \in U$ with $s_0 a_0 s_1 a_1 .. s_n$ being the prefix of a behavior$\}$.*
Given an instance of SSP a policy η is SSP-proper if $P_s^\eta(T_U < \infty) = 1$ for all $s \in S$.
We call η_P the class of SSP-proper policies.
The cost of a policy η at $s \in S$ is defined by $v_s^\eta = E(g(X_{T_U}) + \sum_{k=0}^{T_U - 1} c(X_k, X_{k+1}))$ - X_k is a random variable : its value represents the configuration of distributed system after k computation steps -
The SSP problem is defined by the determination of $v_s^* = inf_{\eta \in \eta_P} v_s^\eta$, for all $s \in S - U$.

We first have to define two assumptions:

Definition 9 (SSP1, SSP2 and L)

- **SSP1** *: There is at least one Markovian SSP-proper policy.*
- **SSP2** *: If η is Markovian and not SSP-proper, then $v_s^\eta = \infty$ for at least one $s \in S$.*
- **L** *: Let (M, U, c, g) be an instance of the SSP problem. We denote $v = (v_s)_{s \in S-U}$ a vector of real numbers. We define the Bellman operator L by*

$$L(v_s) = min_{a \in A(s)} \{c(s,a) + \sum_{t \in S-U} p_{st}(a) v_t + \sum_{t \in U} p_{st}(a) g(t)\}$$

Theorem 1 (Bellman Equations) *If SSP1 and SSP2 are satisfied then the following assertions hold:*

- *The Bellman operator L admits exactly one fixpoint $v^\bullet$ such that $v^\bullet = Lv^\bullet$*
- *$v^\bullet = v^*$*
- *There exists a Markovian policy that reaches the minimum, it suffices to take in each state actions that realize the minimum in the right hand side on the L equation.*

Note that in the initial theorem, we can also obtain the fixpoint by solving a linear programming problem, but we will not to develop this point here.

4.3 Application to self-stabilization

Our goal is to apply the result on the SSP problem to find the convergence time of randomized self-stabilizing algorithms. First,After showing in the previous section how a randomized algorithm can be seen as a Markov decision process, we show now the reduction towards SSP.

Theorem 2 *Computing the convergence time for a randomized self-stabilizing problem can be reduce to an instance of SSP.*

Proof:

- Case 1: Best convergence time. We define U as the set L of legitimate states of the algorithms We define c as identically equal to 1. We define g as identically null.
  With these values of U, c, and g the cost of a policy η is the expectation time of the random variable X_{T_U} under the policy η. The minimum cost of the SSP problem will count the minimum expected number of steps before reaching L under any policy η which is the best convergence time.
- Case 2: Worst convergence time. We define U as the set L of legitimate states of the algorithms We define c as identically equal to -1. We define g as identically null.

With these values of U, c, and g the cost of a policy η is the expectation time of the random variable $-X_{T_U}$ under the policy η. The minimum cost of the SSP problem will count the opposite of the maximum expected number of steps before reaching L under any policy η which is the worst convergence time.

□

Theorem 3 *The instance of SSP obtained from a randomized self-stabilizing problem verifies the assumptions SSP1 and SSP2*

Proof: If the algorithm is self-stabilizing then all the policies allow to converge towards L with probability 1 which ensures that the two assumptions are true, since all the policies are SSP-proper. Note that if the algorithm is only self-stabilizing for a certain class of policies (schedulers) than the reduction to SSP to obtain the worst convergence time satisfies assumptions SSP1 and SSP2. Indeed, SSP1 is true because there exists SSP-proper policies (the one for which the algorithm is self-stabilizing), SSP2 holds because for the non SSP-proper policies the convergence time is infinite so the minimum is $-\infty$. □

Corollary 1 *The best and the worst schedulers (considering the convergence time) for a randomized self-stabilizing algorithm are memoryless and deterministic.*

This comes directly from the theorem.

This last results allows us to only consider memoryless schedulers when studying self-stabilizing algorithms. Indeed, the worst convergence time is always given by a memoryless scheduler, this result will considerably simplify the verification of the correctness and the computation of the complexity of these algorithms.

Moreover, the Bellman equation gives us a way to compute automatically the worst and the best convergence time and to obtain the corresponding schedulers.

We wil now apply all this result to our toy example and to two other algoritms.

Example 3 (Application to the toy example) *Configurations are gathered in class. Configurations in which the tokens are at the same (clockwise) distance d, belong to the same class denoted d. If a token $T1$ is at distance d of the other token $T2$ then $T2$ is at distance N-d of $T1$. Therefore class d and N-d are identical. Notice that if $d = 0$ then the ring has only one token. The best convergence time is easy to guess here, from the configuration of the class $0 < d \leq N/2$ is $2 \times d$, whatever is the ring size. This convergence time is achieved under the following policy: in any configuration, the token at distance d of the other token tries to catch up the unmoving token.*

Let us see how the Bellman equation gives the result.

Let us take $N = 5$, then there are 3 classes c_0, c_1, c_2 There are also three kind of schedulers in each different configuration, one can choose only the closest token (action c), the further (f), or both of them (b). In the corresponding SSP instance we have $U = c_0$ and we can start from the vector $v^0 = (0, 0)$ then applying one time the Bellman operator L we obtain that $v^1 = (1, 1)$ then $v^2 = (3/2, 2)$, $v^3 = (7/4, 11/4)$, $v^4 = (15/8, 26/8)$, $v^5 = (31/16, 57/16)$, $v^6 = (63/32, 120/32)$, $v^7 = (127/64, 247/64)$, $v^8 = (255/128, 502/128)$ and the sequence converges towards $v^\bullet = (2, 4)$ which is effectively a fixpoint. Then, by getting the action in which the minimum is reached we obtained that the best scheduler is the one that chose always action c, that is conform to the intuition.

In the rest of the paper, we will exactly adopt these techniques to find best and worst schedulers (policies) for more difficult examples.

5 Examples

5.1 Self-stabilizing vertex coloring

Algorithm 1 Self-stabilizing vetex coloring algorithm

Variable on p:

c_p color of p node. color takes value in N

Constant in p:

B is a constant having a value in N^*-, we assume that $B > \delta_p$

Action on p:

$\mathcal{R}1$:: $\exists q \in N_p$ such that $c_p = c_q \longrightarrow c_p = random(1, B)$

In this section, we study a very simple self-stabilizing vertex coloring algorithm. The algorithm converges from any configuration to a configuration where neighboring nodes do not have the same color. On a node that has the same color as one of its neighbors $\mathcal{R}1$ is enabled. An enabled node can randomly choose any color in the colors set (i.e. to execute $\mathcal{R}1$ action). All colors have the same probability to be chosen: $1/B$ (B being the color set size). We assume that B is greater than the maximum node degree.

We study the behavior of the self-stabilizing vertex coloring algorithm on a complet network of four nodes. After an analysis of the set of configurations, 4 configurations types was determined:

- All nodes have the same color. This configurations type is named *1 color*. In this type of configurations, all nodes are enabled, four distinct schedule choices exist.
- Every node nodes has a neighbor that has its color, and the configuration has two colors. For instance the configuration (Red, Red, Blue, Blue) belongs to this category. This type is named *2 colors (2)*. In this type of configurations, all nodes are enabled, and 5 distinct schedule choices exist.
- Three nodes have the some color, the last node having a distinct color. For instance the configuration (Red, Red, Red, Blue) belongs to this category. This type is named *2 colors (3)*. In this type of configurations, three nodes are enabled, and three schedule choices exist.
- Only two nodes have the same color. Thus 3 colors exist in a configuration of that type, for instance (Red, Blue, Green, Green). This configurations type is named *3 colors*. In this type of configurations, two nodes are enabled, and two schedule choices exist.

Even on this very simple algorithm, the worst and best strategies are difficult to guess. Thus, without building the Markovien decision process, it is impossible to find the best and worst stabilization time. In the Figure 1, we present the Markov decision processus associated with the self-stabilizing vertex coloring algorithm on a complete network of 4 nodes.

The computation of the best and worst convergence time is done by computing fixpoints of the Bellman operator L. In the Figure 2, we present the best and worst convergence time assuming that $B = 4$, from every configuration types. To compute the worst convergence time, we can start frome the vector $v^0 = (0, 0, 0, 0)$ then applying one time the Bellman operator L we obtain that $v^1 = (-1, -1, -1, -1)$ then $v^2 = (-2, -2, -2, -15/8)$, $v^3 = (-3, -189/64, -188/64, -171/64)$, $v^4 = (253/64, -3937/1024, -975/256, -109/32)$ and converges towards $v^\bullet = (-97/6, -91/6, -89/6, -38/3)$ which is effectively a fixed point.

initial configuration	probability to reach the configuration				
	1 color	2 colors (2)	2 colors (3)	3 colors	4 colors
	3 or 4 nodes perform an action:				
1 color **2 colors (2)** **2 colors (3)**	$1/B^3$	$3(B-1)/B^3$	$4(B-1)/B^3$	$6(B-1)(B-2)/B^2$	$(B-1)(B-2)(B-3)/B^3$
	2 nodes (which does have the same color) perform the action:				
1 color **2 colors (2)**	$1/B^2$	$(B-1)/B^2$	$2(B-1)/B^2$	$(B-1)(B-2)/B^2$	0
	2 nodes (which does have the same color) perform the action:				
2 colors(2) **2 colors (3)** **3 colors**	0	$2/B^2$	$2/B^2$	$5(B-2)/B^2$	$(B-2)(B-3)/B^2$
	1 node performs an action:				
1 color	1/B	0	(B-1)/B	0	0
2 colors (2)	0	1/B	1/B	(B-2)/B	0
2 colors (3)	0	1/B	1/B	(B-2)/B	0
3 colors	0	0	0	3/B	(B-3)/B

Figure 1: Markov Decision Process of vertex coloring algorithm on a complete network of 4 nodes

	Convergence time from				
	1 color	2 colors (2)	2 colors (3)	3 colors	4 colors
Best convergence	34/7	14/3	14/3	4	0
Worst convergence	97/6	91/6	89/6	38/3	0

Figure 2: Convergence time of vertex coloring on 4-node complete network, $B = 4$

The selection of the policy at each iteration of the computation of the fixpoints, helps us to find the best and worst policy on each configuration type. Once the policies known, we could explain why these strategies were optimal.

The best policy or strategy in order to have the fastest convergence time is defined as follows: a node of each color does not move, other nodes try to change their color. Then, after each move, the number of colors does not decrease. The Markov chain defined by the best strategy is given in Figure 3, assuming that $B = 4$.

The worst strategy has two goals: first to maintain as few colours as possible and second to minimize the number of nodes that make a move. For instance, when all nodes have the same color, only one node tries to change its color. ¿From a configuration of type *2 colors (2)*, two nodes having the same color performs a move; with such a policy, it is possible to go back to a configuration with only one color. ¿From a configuration of type *2 colors (3)*, only a node performs the rule action; with such a policy, it is impossible to reach a configuration with four colors, in one step. ¿From a configuration having three colors, two nodes do the rule action; with a such policy, it is possible to reach a configuration with only two colors. With such a strategy, from any illegitimate configuration, the number of colors can decrease in at most two move. The Markov chain defines

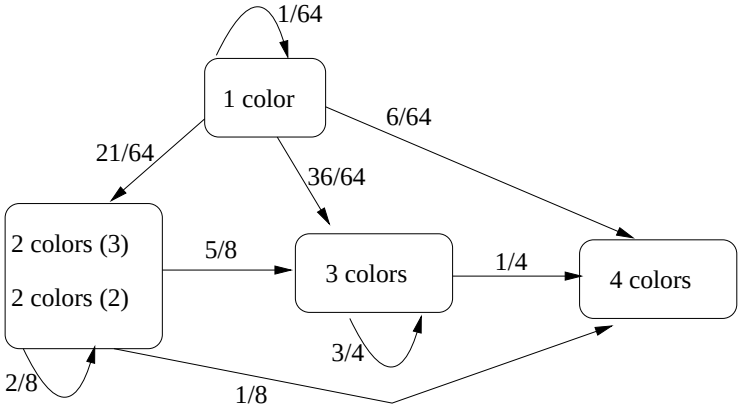

Figure 3: The best strategy for the vertex coloring graph

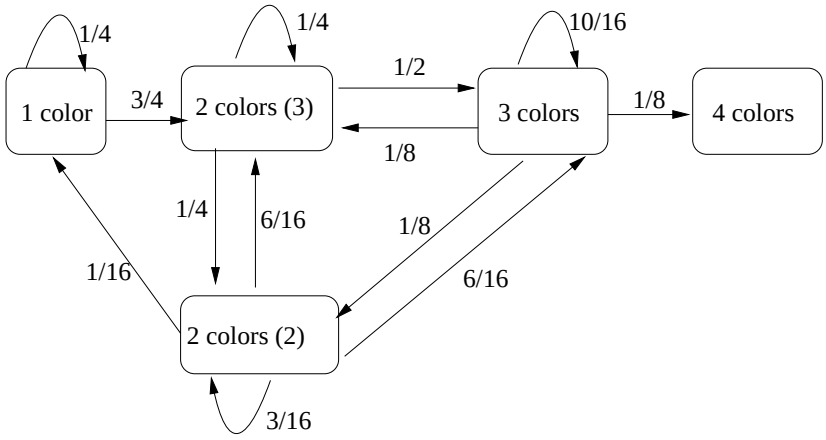

Figure 4: The worst strategy for the vertex coloring graph

by the worst strategy is given in Figure 4, assuming that $B = 4$.

5.2 Self-stabilizing naming algorithms on oriented grids

The problem, here, is to give distinct identifiers to the nodes of a grid network, with the sense of direction (each node knows north, south, west and east).

5.2.1 Deterministic Self-stabilizing algorithm

There are deterministic self-stabilizing algorithms solving this problem. We first present one of them and we compute its stabilization time. Then we randomize the algorithm and we use our techniques to compute the best and the worst stabilization times. We get the (surprising) result that the randomized version is faster than the deterministic one under any scheduler. Here we apply our techniques by hand, what reduces the size of the network we are able to deal with, but their implementation by program would allow to treat more realistic examples. We present one of

these algorithms in Algorithm 2. The identifier is a tuplet of two integers (Val_SW, Val_NW). When every node p verifies the properties (i) and (ii) then the identifiers are distinct:

- (i) the value Val_SW is greater than the values Val_SW of p's South and West neighbors.
- (ii) the value Val_NW is greater than the values Val_NW of p's North and East neighbors.

When a node does not satisfy the property (i) (resp. (ii)) the rule $\mathcal{R}1$ (resp. $\mathcal{R}2$) is enabled. The $\mathcal{R}1$ action sets Val_SW to the smallest value that satisfies the property (i). The $\mathcal{R}2$ action sets Val_NW to the smallest value that satisfies the property (ii).

Algorithm 2 Unique naming self-stabilizing deterministic algorithm

The identifier of p is the couple (Val_SW_p, Val_NW_p).
We name s (resp. w, n, and e) the neighbor of p at the South (resp. West, Nort, and East) of p if such a neighbor exists.

Macro on p:

$Bound\_SW_p = Max(x, y)$. If p has a neighbor at its south then $x = Val\_SW_s$ otherwise $x = 0$. If p has a neighbor at its west then $y = Val\_SW_w$ otherwise $y = 0$.

$Bound\_NW_p = Max(x', y')$. If p has a neighbor at its north then $x' = Val\_NW_n$ otherwise $x' = 0$. If p has a neighbor at its west then $y' = Val\_NW_w$ otherwise $y' = 0$.

Constant in p:

k is a constant having a value in N^+

Action on p:

$\mathcal{R}1$:: if $(Val\_SW_p \leq Bound\_SW_p) \longrightarrow Val\_SW_p := Bound\_SW_p + 1$
$\mathcal{R}2$:: if $(Val\_NW_p \leq Bound\_NW_p) \longrightarrow Val\_NW_p := Bound\_NW_p + 1$

In the following paragraphs, we present the best and worst time to stabilize Val_SW values from IC_N on grid of size $N \times N$.. IC_N is a specific illegitimate configuration defined as follow: the value of Val_SW of a node at distance d of SW is $2(N-1)-d$. In Figure 5, we present such an initial configuration on a grid of size 5×5. ¿From the configuration IC_N, the value Val_SW of each node except the node SW (SW being the node at the far south and far West) has to change in order to get its final value (presented in Figure 5).

Best convergence time Consider the policy that chooses at each computation step, all nodes that can update the value of Val_SW. Note that the Val_SW value of the nodes at distance d of SW is stable after d steps with the deterministic algorithm. On a grid of size $N \times N$, all Val_SW values are stable after $2(N-1)$ computation steps.

Worst convergence time Consider the policy that chooses at each computation step, the farest node of SW that is enabled. Each node p performs at least d_p actions where d_p is the distance of the node to SW. On a grid of size $N \times N$, all Val_SW values are stable after $\sum_p d_p = N^2(N-1)$ computation steps.

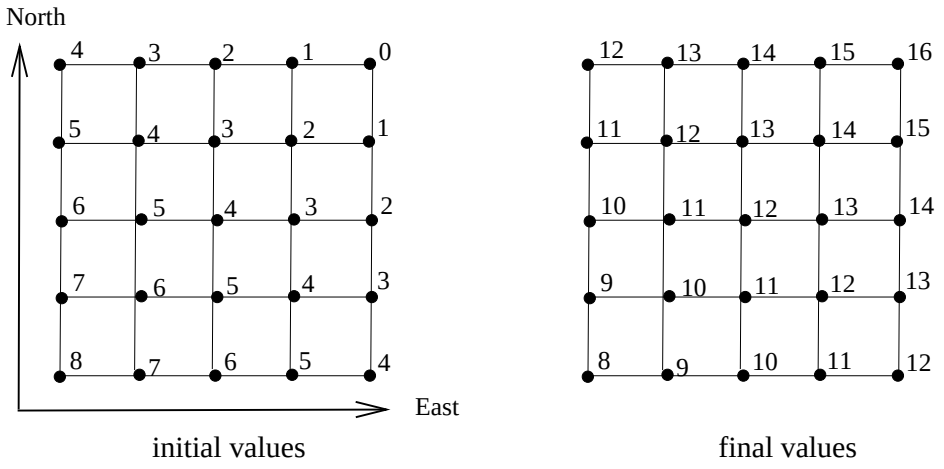

Figure 5: initial and final values of Val_SW

5.2.2 Probabilistic Self-stabilizing algorithm

We present a probabilistic self-stabilizing algorithm based on the same idea as the deterministic algorithm previously presented (algorithm 3). Identifiers are still tuples of two integers (Val_SW, val_NW).

When a node does not verify the property (i) the rule $\mathcal{R}1$ or $\mathcal{R}2$ is enabled. The rule $\mathcal{R}2$ is enabled when the node can update its value Val_SW without forcing one of its neighbors (at its north or at its East) to change its Val_SW value. If such a value does not exist, then $\mathcal{R}1$ is enabled. The $\mathcal{R}2$ action randomly chooses Val_SW among the values which verify (i) and are lesser than the Val_SW value of node's neighbor at East and at North. The $\mathcal{R}1$ action randomly chooses Val_SW among k values - the k smallest values which verify (i).

The rules $\mathcal{R}3$ and $\mathcal{R}4$ changes the Val_NW value. They are similar to the two rules to update Val_SW, respectively $\mathcal{R}1$ and $\mathcal{R}2$.

We study convergence time from IC_N. In IC_N there are $2N-2$ enabled nodes. Note that from a given configuration there are as much different policies as there are subsets of the the set of enabled nodes. When the configuration is symmetrical the policies come by pair. Thus, from IC_N, there are at least 2^{2N-2} distinct policies (schedule choices). Clearly, it is impossible to present on a figure the decision Markovian process on a grid of size bigger than 2×2. Therefore, we present the Markovian decision process on the grid of size 2×2, in Figure 6, assuming that $k > 2$.

To find the best (resp. worst) convergence time, we compute the fixed point of the Bellman operator L when the cost function has the value 1 (resp. -1) in any case. The best convergence time is $2 - \frac{(k-1)(k-2)(2k-3)}{6k^3}$ and the worst convergence time is $4 - \frac{k-2}{k}$. They are better than the same convergence times with the deterministic algorithm respectively 2 computation steps in the best case, and 4 computation steps in the worst case. By extension, we conclude that the probabilistic algorithm is faster than the deterministic algorithm, under any schedule.

Algorithm 3 Unique naming self-stabilizing probabilistic algorithm on oriented grid

The identifier of p is the couple (Val_SW_p, Val_NW_p).
We name s (resp. w, n, and e) the neighbor of p at the South (resp. West, Nort, or East) of p if such a neighbor exists.

Macro on p:

$Lower\_Bound\_SW_p = Max(x, y)$. If p has a neighbor at its south then $x = Val\_SW_s$ otherwise $x = 0$. If p has a neighbor at its west then $y = Val\_SW_w$ otherwise $y = 0$.

$Lower\_Bound\_NW_p = Max(x', y')$. If p has a neighbor at its north then $x' = Val\_NW_n$ otherwise $x' = 0$. If p has a neighbor at its west then $y' = Val\_NW_w$ otherwise $y' = 0$.

$Upper\_Bound\_SW_p = Min(z, w)$. If p has a neighbor at its north then $z = Val\_SW_n$ otherwise $z = 0$. If p has a neighbor at its east then $w = Val\_SW_e$ otherwise $w = 0$.

$Upper\_Bound\_NW_p = Min(z', w')$. If p has a neighbor at its south then $z' = Val\_NW_s$ otherwise $z = 0$. If p has a neighbor at its east then $w' = Val\_NW_e$ otherwise $w = 0$.

Constant in p:

k is a constant having a value in N^*

Action on p:

$\mathcal{R}1$:: if $(Val\_SW_p \leq Lower\_Bound\_SW_p) \wedge (Lower\_Bound\_SW_p + 1 \geq Upper\_Bound\_SW_p) \longrightarrow$
$Val\_SW_p := random(Lower\_Bound\_SW_p + 1..Lower\_Bound\_SW_p + k)$;

$\mathcal{R}2$:: if $(Val\_SW_p \leq Lower\_Bound\_SW_p) \wedge (Lower\_Bound\_SW_p + 1 < Upper\_Bound\_SW_p) \longrightarrow$
$Val\_SW_p := random(Lower\_Bound\_SW_p + 1..Upper\_Bound\_SW_p - 1)$;

$\mathcal{R}3$:: if $(Val\_NW_p \leq Lower\_Bound\_NW_p) \wedge (Lower\_Bound\_NW_p + 1 \geq Upper\_Bound\_NW_p)$
$\longrightarrow Val\_NW_p := random(Lower\_Bound\_NW_p + 1..Lower\_Bound\_NW_p + k)$;

$\mathcal{R}4$:: if $(Val\_NW_p \leq Lower\_Bound\_NW_p) \wedge (Lower\_Bound\_NW_p + 1 < Upper\_Bound\_NW_p)$
$\longrightarrow Val\_NW_p := random(Lower\_Bound\_NW_p + 1..Upper\_Bound\_NW_p - 1)$;

6 Conclusion

In this paper, we show that concerning the convergence time for probabilistic self-stabilizing algorithm, we can reduce the field of schedulers to those which are deterministic and memoryless.
To prove this result we use a reduction towards a well-studied problem (Stochastic Shortest Path). By the way, this reduction gives us a way to compute automatically both the convergence time and the worst and best schedulers.
We need sometimes to find the best and the worst "'fair schedulers"' , we conjecture that under weak fairness assumption (the measure of non-fair execution is null) the results hold. More, we show that computations can be made by hand only for very small networks and prove the necessity to implement them by program.

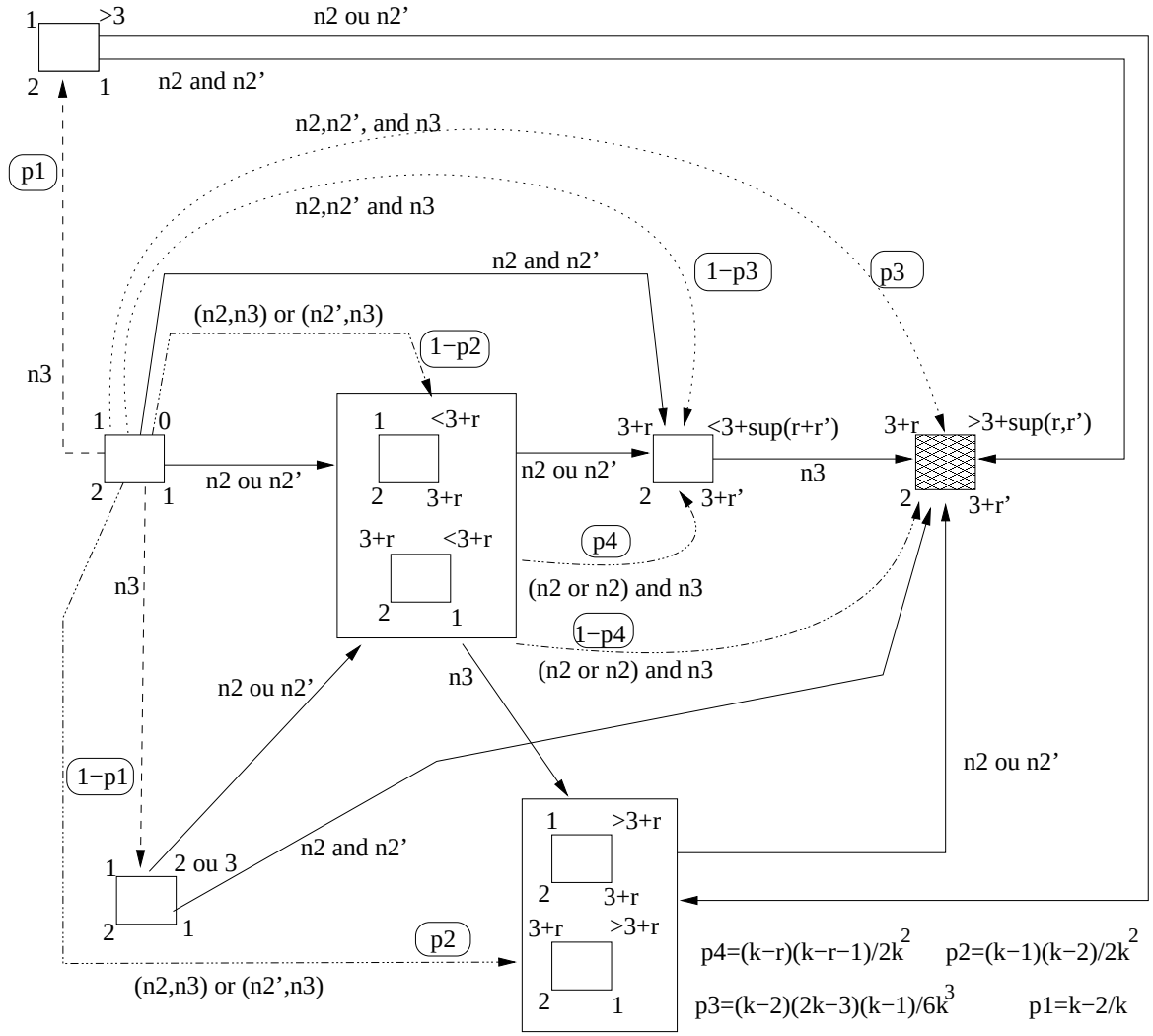

Figure 6: The Markov decision process to set Val_SW values on grid 2×2, where $k > 2$

References

[1] J. Beauquier, S. Cordier, and S. Delaët. Optimum probabilistic self-stabilization on uniform rings. In *WSS95 Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 15.1–15.15, 1995.

[2] D.P. Bertsekas and J.N. Tsitsiklis. An analysis of stochastic shortest path problems. *Math of Op. Res.*, 16(2):580–595, 1991.

[3] L. de Alfaro. *Formal Verification of Probabilistic systems.* PhD Thesis, Stanford University, 1997.

[4] S. Dolev, A. Israeli, and S. Moran. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Software Engineering*, 21:429–439, 1995.

[5] L. Fribourg and S. Messika. Brief announcement: Coupling for markov decision processes - application to self-stabilization with arbitrary schedulers. In *PODC05 Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, page 322, 2005.

[6] L. Fribourg, S. Messika, and C. Picaronny. Coupling and Self-stabilization. In *Proc. 18th Int. Conf. on Distributed Computing (DISC 2004), LNCS 3274*, pages 201–215. Springer, 2004.

[7] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, 1990.

[8] D. Lehmann and M. O. Rabin. On the advantages of free choice: a symmetric and fully-distributed solution to the dining philosophers problem. In *Proc. 8th Annual ACM Symp. on Principles of Programming Languages (POPL'81)*, pages 133–138, 1981.

[9] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, Jan. 1986.

[10] A. Pogosyants and R. Segala. Formal verification of timed properties of randomized distributed algorithms. In *PODC95 Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 174–183, 1995.

[11] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized concensus algorithm of aspnes and herlihy: a case study. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320*, pages 22–36, 1997.

[12] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In *CONCUR'94 Fifth International Conference on Concurrency Theory, Springer-Verlag LNCS:836*, pages 481–496, 1994.

[13] M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS'85*, pages 327–338, 1985.

A Complete Algorithm of Toy example

Algorithm 4 token circulation on anonymous and unidirectional rings

Constant:
N is the ring size. m_N is the smallest integer not dividing N.
Field variables on p: v_p is a variable taking value in [0, m_N -1].
Random Variables on p:
$rand\_bool_p$ taking value in {1, 0}. Each value has a probability 1/2.
Predicate on p:
$Token_p \equiv v_p - v_{lp} \neq 1 \bmod m_N$
Macro on p:
$v_p := (v_{lp} + 1) \bmod m_N$;

Action on p:
$\mathcal{A}$:: $Token_p \rightarrow$ if $(random\_bool_p = 0)$ then $Pass\_Token_p$.