International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 2, Number 2, pages 147–159, July 2012

Probabilistic Self-Stabilization and Biased Random Walks on Dynamic Graphs

Masafumi Yamashita[1]
Department of Informatics, Kyushu University
Motooka, Nishi, Fukuoka 819-0395 Japan.
mak@inf.kyushu-u.ac.jp

Received: May 1, 2012
Revised: June 1, 2012
Accepted: June 29, 2012
Communicated by Koji Nakano

Abstract

A distributed system is said to be probabilistic self-stabilizing, if it eventually converges to a legitimate computation with probability 1, starting from any global configuration. Like a self-stabilizing system, a probabilistic self-stabilizing system tolerates any number of transient failures and recovers a legitimate computation, but only probabilistically, unlike a self-stabilizing system, which recovers it deterministically even in the *worst* case. However, a self-stabilizing algorithm is in general difficult to design and even impossible for some problems. A probabilistic self-stabilizing, on the other hand, is easier to design. To see this, we discuss how a probabilistic self-stabilizing system is constructible from a given weak stabilizing system, which can recover a legitimate computation only in the *best* case.

An execution of a probabilistic self-stabilizing system can be modeled by a random walk on a graph, and its performance can be evaluated in terms of some quantities, e.g., the hitting and the cover times, of the corresponding random walk. The hitting and the cover times of random walks have been studied extensively, but most of them consider standard (i.e., unbiased) random walks on static graphs. We discuss how to design biased random walks whose hitting and cover times are faster than standard random walks, to improve the performance of probabilistic self-stabilizing system. We also discuss random walks on dynamic graphs to analyze a probabilistic self-stabilizing system such that its communication network topology frequently changes.

Keywords: self-stabilization, probabilistic self-stabilization, weak stabilization, Markov chain, random walk, hitting time, cover time, biased walk, dynamic graph

1 Introduction

The aim of this paper is to discuss how to construct efficient probabilistic self-stabilizing systems.

A distributed system is said to be *self-stabilizing*, if it eventually converges to legitimate computation starting from an arbitrary initial (global) configuration [12]. An obvious advantage of a self-stabilizing system is that initialization is unnecessary. However, a more important one is that it can tolerate any finite number of transient failures, each of which can arbitrarily modify the current

[1]This work was supported in part by a Grant-in-Aid for Scientific Research from Japan Society for the Promotion Science.

configuration, since it will eventually start legitimate computation, regarding the current illegitimate configuration reached as the result of a transient error as an initial configuration. In order to formally discuss stabilization, let us start with introducing the model of distributed system.

1.1 Distributed Systems

A distributed system consists of n communicating processes. As usual, we model its communication network by a digraph $N = (P, L)$, where P and L respectively represent the sets of the processes and the uni-directional communication links, i.e., an edge (p, q) is in L if and only if process q can directly obtain information from process p. Communication is carried out by means of *shared variables.* Each process p holds a set of shared variables and can access them to read and to write. It can also access the variables of its predecessors to read their values.

A variable in a process takes a value from a pre-determined domain. The *state of a process* is a function that assigns, to each of the variables in the process, a value in its domain. Let S_p be the set of all states of a process $p \in P$. Then the Cartesian product $\mathcal{C} = \Pi_{p \in V} S_p$ of S_p for all $p \in P$ forms the set of (global) *configurations.* We assume that the read and write operations are *atomic* and finish at a time even when the read operation is initiated by a neighbor. A process p can change its state by executing its *algorithm* $\mathcal{A}$, which is described by a sequence of guarded actions, where the guard of an action in p is a boolean expression that involves some variables of p and its predecessors (of N). An action of some process p is enabled in a configuration γ when its guard is satisfied. We say that p is *enabled* in γ if one of its actions is enabled in γ. Simultaneous executions of actions by a set of processes induce a transition between two configurations. If a configuration γ' yields from a configuration γ, we denote this transition by $\gamma \mapsto \gamma'$. For any configuration γ, $\gamma \mapsto \gamma$ holds, which corresponds to the result of “simultaneous executions” by an empty set of processes.

All potential behaviors of a distributed system executing $\mathcal{A}$ on N is described by a digraph $\mathcal{S} = (\mathcal{C}, \mapsto)$, which models a *distributed system.* An *execution* of $\mathcal{S}$ is a possibly infinite (directed) path $\gamma_0, \gamma_1, \ldots$ of $\mathcal{S}$, starting from a configuration in $\mathcal{C}$. We say that a configuration γ' is *reachable* from a configuration γ, if there is a path from γ to γ' in $\mathcal{S}$.

There are in general more than one execution $\mathcal{E} = \gamma_0, \gamma_1, \ldots$ in $\mathcal{S}$ for some initial configuration $\gamma_0 \in \mathcal{C}$. A *scheduler* determines which executions among them are executable by non-deterministically choosing some processes to *activate*, i.e., to have them execute the corresponding actions, from the processes enabled in γ_t, at each time instant t. We denote by $\mathcal{E}(\mathcal{S}, \sigma)$ the set of all executions that can occur as an execution of $\mathcal{S}$ under a scheduler σ. The scheduler of a distributed system is an abstraction of the given environment in which the distributed system is executed, and thus we cannot choose (or control) it.

A scheduler is *proper*, if it always activates at least one process as long as there are enabled processes. A scheduler is *weakly fair* if every *continuously* enabled process is eventually activated. A scheduler is *strongly fair* if every process that is enabled *infinitely often* is eventually activated. A strongly fair scheduler is thus weakly fair by definition. Note that we do not request a weakly fair (and hence a strongly fair) scheduler to be proper. The *synchronous* scheduler always activates every enabled process. The synchronous scheduler is proper and strongly fair. Unlike the synchronous scheduler, there are many different weakly (resp. strongly) fair schedulers. In the following, when we say the scheduler is weakly (resp. strongly) fair, we mean the weakly (resp. strongly) fair scheduler which can produce every weakly (resp. strongly) fair execution.

A distributed system is constructed to solve some problem. Let $\mathcal{SP}$ be the set of ”correct” executions in $\mathcal{S}$, which *specifies* the problem it solves. For example, the purpose of a token ring system is to circulate a single token in a ring. When the system works correctly, its execution circulates the token in the ring, and $\mathcal{SP}$ of the system is the set of such correct executions.

1.2 Self-Stabilization and Weak Stabilization

Let $\mathcal{S}, \sigma$ and $\mathcal{SP}$ be a distributed system, a scheduler and a specification, respectively.

Definition 1 (Self-Stabilization [12]). *$\mathcal{S}$ is* self-stabiliz- ing *for $\mathcal{SP}$ under σ, if there is a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ satisfying:*

- **Strong Closure Property:** *Every execution in $\mathcal{E}(\mathcal{S}, \sigma)$ with an initial configuration in $\mathcal{L}$ belongs to $\mathcal{SP}$.*

- **Certain Convergence Property:** *Every execution in $\mathcal{E}(\mathcal{S}, \sigma)$ eventually reaches a configuration in $\mathcal{L}$.*

Definition 2 (Weak Stabilization [15])**.** *$\mathcal{S}$ is* weak stabilizing *for $\mathcal{SP}$ under σ, if there exists a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ satisfying:*

- **Strong Closure Property:** *Every execution in $\mathcal{E}(\mathcal{S}, \sigma)$ with an initial configuration in $\mathcal{L}$ belongs to $\mathcal{SP}$.*

- **Possible Convergence Property:** *For any configuration $\gamma \in \mathcal{C}$, there is an execution in $\mathcal{E}(\mathcal{S}, \sigma)$ that starts at γ and eventually reaches a configuration in $\mathcal{L}$.*

A configuration is said to be *legitimate* if it is in $\mathcal{L}$. Let σ_{WF} (resp. σ_{SF}) denote the weakly (resp. strongly) fair scheduler. Some simple properties are:

Property 1. *1. For any distributed system $\mathcal{S}$ and specification $\mathcal{SP}$, if $\mathcal{S}$ is* self-stabilizing *for $\mathcal{SP}$ under σ_{WF}, then it is self-stabilizing for $\mathcal{SP}$ under σ_{SF}.*

2. For any distributed system $\mathcal{S}$ and specification $\mathcal{SP}$, if $\mathcal{S}$ is weak stabilizing *for $\mathcal{SP}$ under σ_{SF}, then it is weak stabilizing for $\mathcal{SP}$ under σ_{WF}.*

3. For any distributed system $\mathcal{S}$, specification $\mathcal{SP}$ and scheduler σ, if $\mathcal{S}$ is self-stabilizing *for $\mathcal{SP}$ under σ, then it is* weak stabilizing *for $\mathcal{SP}$ under σ.*

In fact, weak stabilization is strictly weaker than self-stabilization, since there is a problem that has a weak stabilizing algorithm but has no self-stabilizing algorithm [10]. The next theorem relates the weak and the self-stabilization.

Theorem 1 ([15])**.** *Let $\mathcal{S}$, $\mathcal{SP}$ and σ be a distributed system, a specification for $\mathcal{S}$, and a scheduler, respectively. If $\mathcal{S}$ is weak stabilizing for $\mathcal{SP}$ under σ, then $\mathcal{S}$ is self-stabilizing for $\mathcal{SP}$ under σ, provided that*

1. *$\mathcal{S}$ has a finite number of configurations, and*

2. *σ satisfies the* Gouda's strong fairness *condition: For any transition $\gamma \mapsto \gamma'$, if γ occurs infinitely often in an execution $\mathcal{E}$, then $\gamma \mapsto \gamma'$ also appears infinitely often in $\mathcal{E}$.*

Since the weak stabilization is a strictly weaker concept than the self-stabilization, we can expect that the Gouda's strong fairness is strictly stronger than the strong fairness. Indeed, we have:

Theorem 2 ([10])**.** *Suppose that the system has a finite number of configurations. The Gouda's strong fairness is strictly stronger than the strong fairness.*

Based on Theorem 1, Gouda suggested to design a self-stabilizing system as a weak stabilizing system, to relax the difficulty of designing a self-stabilizing system, which method however leaves a formidable problem of forcing the given environment that the scheduler abstracts to observe the Gouda's strong fairness (besides the strongly fairness).

We will define the concept of probabilistic self-stabilization [17, 20] in Section II and argue that this notion may close the gap between the weak and the self-stabilization. Then we discuss how to design and analyze probabilistic self-stabilizing systems in Section III. We will discuss biased random walks on dynamic graphs as a possible tool to this end, despite that most of the studies on random walks have considered unbiased random walks on static graphs.

2 Probabilistic Self-Stabilization

We now introduce the concept of probabilistic self-stabilization [17, 20, 10]. A probabilistic scheduler chooses and activates some of the enabled processes probabilistically.

Definition 3 (Probabilistic Self-Stabilization [17, 20])**.** *A (possibly randomized) distributed system $\mathcal{S}$ is* probabilistically self-stabilizing *for a specification $\mathcal{SP}$ under (possibly probabilistic) scheduler σ, if there is a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ satisfying:*

- **Strong Closure Property:** *Every execution in $\mathcal{E}(\mathcal{S}, \sigma)$ with an initial configuration in $\mathcal{L}$ belongs to $\mathcal{SP}$.*
- **Probabilistic Convergence Property:** *Every execution in $\mathcal{E}(\mathcal{S}, \sigma)$ eventually reaches a configuration in $\mathcal{L}$ with probability 1.*

The concept of probabilistic self-stabilization was first introduced in [17, 20], in which the authors proposed randomized algorithms that work as a probabilistic self-stabilizing system under a non-deterministic scheduler. Later in [10], the authors introduced a randomized scheduler, modeling the environment as a stochastic process, and discuss deterministic distributed systems that work as a probabilistic self-stabilizing system under the randomized scheduler. Note again that a scheduler is an abstraction of the environment, so we cannot design a favorable probability distribution that the scheduler uses to select processes from enabled ones to activate them, like the case of non-deterministic scheduler.[2] We discuss this latter case in Subsection A, and leave the former case in Subsection B. These two subsections make the differences of the two cases clear. We then give examples of probabilistic self-stabilizing systems in Subsection C, and in Subsection D, we relate analyses of probabilistic self-stabilizing systems with random walks.

2.1 Probabilistic Self-Stabilization under Randomized Scheduler

We consider a *randomized scheduler* that always chooses each of the enabled processes at random with probability 1/2. We denote the randomized scheduler by σ_R. We however can modify the definition of randomized scheduler so that it always chooses each of the enabled processes at random with any probability in (ϵ_L, ϵ_H), where $0 < \epsilon_L \leq \epsilon_H < 1$ are arbitrarily chosen positive constants, and essentially the same results can be obtained. Note that the randomized scheduler is not proper, since it may not activate any process even if there are enabled ones.

Let $\mathcal{S}$ be a weak stabilizing system for $\mathcal{SP}$ under σ_{SF}, and consider $\mathcal{S}$ under σ_R. Since $\mathcal{S}$ is weak stabilizing under σ_{SF}, for any $\gamma \in \mathcal{C}$ there is a legitimate configuration $\gamma' \in \mathcal{L}$ reachable from γ in $\mathcal{S}$. On the other hand, if there is a (finite) path from γ to γ' in $\mathcal{S}$, then the path can occur as a prefix of an execution in σ_{SF}, since a finite path never violate the strongly fairness.

Property 2. *$\mathcal{S}$ is weak stabilizing for $\mathcal{SP}$ under σ_{SF}, if and only if for any $\gamma \in \mathcal{C}$, there is a legitimate configuration $\gamma' \in \mathcal{L}$ reachable from γ in $\mathcal{S}$.*

Suppose that $\mathcal{S}$ under σ_R is in a configuration γ and let $U \subseteq V$ be the set of enabled processes in γ. Then a subset $W \subseteq U$ is selected uniformly at random from 2^U, and the next configuration is determined from W. We model $\mathcal{S}$ under σ_R as a (possibly infinite state) Markov chain $\mathcal{M}$ defined over set $\mathcal{C}$ of "states" associated with the transition probability matrix $\mathbf{P} = (p_{\gamma\gamma'})_{\gamma,\gamma' \in \mathcal{C}}$, where the transition probability $p_{\gamma\gamma'}$ from γ to γ' is $2^{-|U|}$ by the definition of σ_R. We call $\mathcal{M}$ the Markov chain corresponding to $\mathcal{S}$ under σ_R.

Recall that a (finite or infinite) Markov chain is called *irreducible* if its state transition graph is strongly connected. A state γ is called *recurrent* if every evolution $x_0(= \gamma), x_1, \ldots$ that starts at γ returns γ with probability 1, and otherwise it is called *transient.*

[2]We in this paper assume that the probability distribution of a randomized algorithm is a part of design issue, but the probability distribution of a randomized scheduler is a constraint for the design of a probabilistic self-stabilizing system. However, depending on applications, the environment may be adjustable, in which case the probability distribution of a randomized scheduler also becomes a variable to be optimized.

Consider first the case in which the algorithm $\mathcal{A}$ is finite state, i.e., the domain of each of the variables is finite. Then the number of configurations is finite and the corresponding Markov chain $\mathcal{M}$ is also finite, since a distributed system consists of a finite number of processes.

Theorem 3 ([10])**.** *Let $\mathcal{S}$ be the distributed system executing a finite state algorithm $\mathcal{A}$ on a communication network N, and let $\mathcal{SP}$ be any specification for $\mathcal{S}$. Then $\mathcal{S}$ is weak stabilizing for $\mathcal{SP}$ under σ_{SF} if and only if it is probabilistically self-stabilizing for $\mathcal{SP}$ under σ_R.*

Proof. If part is obvious. An outline of *Only-If* part: Consider the strongly connected component graph $H = (\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k\}, A)$ of $\mathcal{S}$. H is a finite DAG. If $\mathcal{C}_i$ is a sink of H, $\mathcal{C}_i$ contains a legitimate configuration γ_i. It is easy to observe that any evolution $x_0, x_1, \ldots$ of $\mathcal{M}$ eventually enters a sink $\mathcal{C}_i$ and then reaches γ_i with probability 1, since a configuration is recurrent if and only if it belongs to a sink. The proof completes by Property 2. □

Next consider the case in which $\mathcal{A}$ is infinite state. Like the finite state case, $\mathcal{S}$ holds the strong closure property for a specification $\mathcal{SP}$ under σ_{SF}, if and only if it holds the same property for $\mathcal{SP}$ under σ_R. Also, $\mathcal{S}$ holds the possible convergence property holds for $\mathcal{SP}$ under σ_{SF}, if it holds the probabilistic convergence property for $\mathcal{SP}$ under σ_R. However, its converse does not hold in general, unlike the case of finite state algorithms.

Consider a distributed system consisting of three processes X, Y and Z, on which an algorithm $\mathcal{A}$ is executed. The processes provide a variable v, whose domain is the set of non-negative integers, and $\mathcal{A}$ consists of a single guarded action to increment v, where its guard is **true**. All processes are thus always enabled. We call a configuration legitimate if $v_X + v_Y = v_Z$, where v_p denotes the variable v in process p, and let $\mathcal{SP}$ be the set of executions $\mathcal{E}$ that contains a legitimate configuration. Then this distributed system holds the possible convergence property for $\mathcal{SP}$ under σ_{SF}, but does not hold the probabilistic convergence $\mathcal{SP}$ under σ_R, since the corresponding Markov chain $\mathcal{M}$ is an asymmetric unrestricted random walk on the line, which is known to be irreducible and transient [7, Example 8.9].

Theorem 4 ([11])**.** *There is an infinite state algorithm $\mathcal{A}$ that is weak stabilizing for a specification $\mathcal{SP}$ under σ_{SF}, such that $\mathcal{A}$ is not probabilistically self-stabilizing for $\mathcal{SP}$ under σ_R.*

Thus we cannot fully generalize Theorem 3 to the case of infinite state algorithms. Ref. [11] shows some sufficient conditions; the following is one of them.

Theorem 5 ([11])**.** *Let $\mathcal{S}$ and $\mathcal{M}$ be an infinite state distributed system and the infinite Markov chain corresponding to $\mathcal{S}$, respectively. Assume that $\mathcal{M}$ is (irreducible and) recurrent. Let $\mathcal{SP}$ be any specification for $\mathcal{S}$. Then $\mathcal{S}$ is weak stabilizing for $\mathcal{SP}$ under σ_{SF}, if and only if it is probabilistically self-stabilizing for $\mathcal{SP}$ under σ_R.*

Note that we can replace the scheduler from σ_{SF} to σ_{WF} in the results of this subsection by Property 1(2).

2.2 Probabilistic Self-Stabilization with Randomized Algorithm

As explained at the top of this section, the randomized scheduler is not the only way to introduce the randomness to the behavior of a distributed system. We in this subsection consider a randomization of algorithm. Let $\mathcal{A}$ be a deterministic algorithm and consider a simple transformer that transforms $\mathcal{A}$ into a randomized algorithm $\mathcal{A}^*$ as follows: Whenever an enabled process is activated, it first tosses a coin and then executes the action, if and only if it gets the "head". Formally, in $\mathcal{A}^*$, in addition to the variables used in $\mathcal{A}$, we provide a new boolean variable B. We then replace each action "*guard* $\longrightarrow$ *statement*" of $\mathcal{A}$ into

$$\textit{guard} \longrightarrow\ B := \textit{Rand}();\ \textbf{if}\ B\ \textbf{then}\ \textit{statement},$$

where *Rand* is a random bit generator, which returns one of the boolean values with the same probability. You might expect a similar result to Theorem 3. Suppose that there is a distributed system $\mathcal{S}$ that executes a finite state algorithm $\mathcal{A}$ on a communication network N, and let $\mathcal{SP}$ be a specification for $\mathcal{S}$. Let us compare $\mathcal{S}$ with the randomized distributed system $\mathcal{S}^*$ that executes randomized algorithm $\mathcal{A}^*$ on N, which we expect to solve the "same" problem as $\mathcal{S}$. To this end, we need to prepare a specification $\mathcal{SP}^*$ for $\mathcal{S}^*$, which must be different from $\mathcal{SP}$, since $\mathcal{A}^*$ uses variables B that are not used in $\mathcal{A}$ and the configuration space $\mathcal{C}^*$ of $\mathcal{S}^*$ is different from $\mathcal{C}$ of $\mathcal{S}$. We define $\mathcal{SP}^*$ in such the way that $\mathcal{E}^* \in \mathcal{SP}^*$ if and only if the execution $\mathcal{E}$ of $\mathcal{S}$ constructed from $\mathcal{E}^*$ by removing all value information on these variables B from every configuration in $\mathcal{E}^*$ is in $\mathcal{SP}$; intuitively, $\mathcal{SP}^*$ is the same as $\mathcal{SP}$ when ignoring these variables.

Theorem 6. *There is a distributed system $\mathcal{S}$ that executes a finite state algorithm $\mathcal{A}$ on a communication network N and a specification $\mathcal{SP}$ such that $\mathcal{S}$ is weak stabilizing for $\mathcal{SP}$ under σ_{SF}, but the distributed system $\mathcal{S}^*$ that executes $\mathcal{A}^*$ on N is not probabilistic self-stabilizing for $\mathcal{SP}^*$ under σ_{SF}.*

Proof. An outline: Consider a network N of two processes X and Y such that there is a bidirectional link between them. Algorithm $\mathcal{A}$ uses a variable v whose domain is $\{0, 1\}$, and consists of a single guarded action. Its guard is **true** and the action is: If $(v_X, v_Y) = (0, 0)$ or $(1, 1)$ then $v := 1$, else if $(v_X, v_Y) = (0, 1)$ or $(1, 0)$ then $v := 0$. Call $(1, 1)$ legitimate, and define $\mathcal{SP}$ by the set of all executions containing the legitimate configuration. Obviously it is weak stabilizing under σ_{SF} (since there is an execution that leads to the legitimate configuration $(1, 1)$).

Observe that the legitimate configuration cannot be reached unless more than one process is activated simultaneously at $(0, 0)$. Since σ_{SF} can always activate at most one process, $\mathcal{S}^*$ is not probabilistic self-stabilizing under σ_{SF}. □

However we can show the following theorem.

Theorem 7 ([10])**.** *Suppose that $\mathcal{S}$ is finite state. $\mathcal{S}$ is probabilistically self-stabilizing for $\mathcal{SP}$ under σ_R, if and only if $\mathcal{S}^*$ is probabilistically self-stabilizing for $\mathcal{SP}^*$ under the synchronous scheduler.*

Corollary 1 ([10])**.** *Let $\mathcal{S}$ be the distributed system executing a finite state algorithm $\mathcal{A}$ on a communication network N, and let $\mathcal{SP}$ be any specification for $\mathcal{S}$. Then $\mathcal{S}$ is weak stabilizing for $\mathcal{SP}$ under σ_{SF} if and only if $\mathcal{S}^*$ is probabilistically self-stabilizing for $\mathcal{SP}^*$ under the synchronous scheduler.*

So far we introduced two types of probabilistic self-stabilizing systems. In the case of deterministic systems under a randomized scheduler, although Theorem 3 holds for a variety of randomized schedulers (not only σ_R), we cannot take advantage of tuning the probability distribution of scheduler to improve the performance of the system. On the other hand, in the case of randomized systems under the synchronous scheduler, optimizing the probability distribution of the algorithm is an important design issue to improve the performance of the system (provided that the environment can be modeled by the synchronous scheduler). These two randomnesses thus have different nature.

2.3 Examples of Probabilistic Self-Stabilization

(1) *Probabilistic Self-Stabilization under Probabilistic Scheduler*:

The token circulation problem for a uni-directional ring is known as the first problem for which a self-stabilizing algorithm is given [12]. His solution is not uniform; it has turned out that there is a uniform (deterministic) solution (or more formally solution for anonymous ring), if and only if the size of a ring is prime [6, 17].

Consider an anonymous uni-directional ring $R_n = (P, L)$, where $P = \{0 \leq i \leq n-1\}$ and $L = \{(i, i+1) : i = 0, 1, \ldots, n-1\}$, provided that the process name i is calculated modulo n. We design a probabilistic self-stabilizing token circulation algorithm for R_n, where $n \geq 3$ is an arbitrary number, by introducing the randomness in the scheduler.

A basic trick used in [6] for designing their algorithm is as follows: It uses a register v whose domain is $D = \{0, 1, \ldots, n-2\}$. We define that a process i has a token if $v_i \neq v_{i-1} + 1$, and i is enabled whenever it has a token, where v_i denotes the register v of process i. There is at least one token at any configuration since $|D| = n - 1$. A process i, which is enabled, executes $v_i := v_{i-1} + 1$

when it is activated, and then loses the token by definition. Confirm that the algorithm is uniform and i can execute it without knowing i.

The activation of i does not increase the number of tokens; the token of i disappears and a new token is created in $i+1$ only when there was no token in $i+1$ before v_i is updated. Informally, the activation of i transfers the token from i to $i+1$ if there is no token in $i+1$; otherwise if there is a token in $i+1$, then one of them disappears. It is thus obvious to find an activation sequence that decrements the number of tokens in R_n when at least two tokens exist, which shows that this trick alone works as a weak stabilizing algorithm under σ_{SF}, or equivalently as a probabilistic self-stabilizing algorithm under σ_R, for *all* n (not only for prime numbers).

(2) *Randomized Algorithms for Probabilistic Self-Stabilization*:

We next give randomized algorithms for probabilistic self-stabilization.

For any uni-directional ring of an odd size, Herman [17] presented a randomized algorithm to realize a probabilistic self-stabilizing token circulation system under the synchronous scheduler. Using the notations above, let $\{0,1\}$ be the domain of register v, and define that i has a token if and only if $v_i = v_{i-1}$. His algorithm updates v_i to v_{i-1} if $v_i \neq v_{i-1}$, and v_i is set to a random bit, otherwise.

An outline of Israeli and Jalfon's algorithm [20] that works on *any* communication network $N=(P,L)$ (not only a uni-directional ring) is as follows: Each process $i \in P$ provides a register v_i whose domain is the set of integers. When $v_i \geq \max_{j \in N(i)} v_j$, i is enabled and has a token, where $N(i)$ is the set of neighbors of i in N. Then there is at least one token in any configuration. A process i, which is enabled, select a process j in $N(i)$ uniformly at random, and have j sufficiently increase v_j so that $v_j \geq \max_{k \in N(j)} v_k$ is satisfied, i.e., to transfer the token to j. Then tokens make random walks on the graph and they coalesce when they meet at a vertex.

A random walk which selects each of the neighbors with the same probability as the next vertex like in this algorithm is called the *standard random walk*. Since two standard random walks eventual meet with probability 1, the system converges to a legitimate configuration with probability 1, and the standard random walk eventually visits all the vertices (e.g., [1]). It is worth emphasizing that their algorithm is probabilistic self-stabilizing even under an unfair scheduler.

2.4 Probabilistic Self-Stabilization and Random Walks

In Section II, we have introduced two types of probabilistic self-stabilizing systems and discussed how to design them via weak stabilizing systems. We view the scheduler as the environment so that we cannot choose (or design) it, but we of course can design a randomized algorithm so that the probabilistic self-stabilizing system can show the best performance. Taking the Israeli and Jalfon's algorithm [20] in Subsection C as an example, we discuss random walks as a tool to analyze probabilistic self-stabilizing systems.

The Israeli and Jalfon's algorithm makes use of the standard random walk on a finite graph $G=(V,E)$ (i.e., communication network). Some of the performance measures that we are interested in are (1) the convergence time - how long before only one token remains, (2) the waiting time - how long before receiving a token, (3) the cover time - how long before a token visiting all vertices, and (4) the fairness - how fair the token circulation is.

The *hitting time* $H_G(u,v)$ from u to v is the expected number of steps necessary to reach v from u, and the (maximum) hitting time H_G of G is defined as $H_G = \max_{u,v \in V} H_G(u,v)$. The *cover time* $C_G(u)$ from u is the expected number of steps necessary to visit all vertices from u, and the (maximum) cover time C_G of G is defined as $C_G = \max_{u \in V} C_G(u)$. Then the convergence, the waiting and the cover times of the Israeli and Jalfon's algorithm on a communication graph G are evaluated by the hitting and the cover times of the standard random walk on G.

The standard random walk has been extensively studied (see e.g., [1]). Since the hitting and the cover times of standard random walk on any graph of order n is $O(n^3)$, and there is a graph called a Lollipop graph, for which the hitting and the cover times of standard random walk is $\Omega(n^3)$ [13, 14], rough bounds on the convergence, the waiting and the cover times are all $\Theta(n^3)$.

You might think that this bound $\Theta(n^3)$ is by no means good enough. It is thus natural to consider a "biased" random walk, in which a vertex u can select a one from its neighbors v with different probabilities. Indeed, this idea works in an extremal case in which G is used to define the transition probability from u to v. But obviously such an algorithm is not practical, since knowing G is in general a very expensive task, and even impossible frequently. Thus the real question here is to understand the impact of local information to reduce the hitting and the cover times.

The fairness is analyzed by using the stationary distribution of the standard random walk. Since the stationary distribution of a vertex u whose degree is $d(u)$ is $d(u)/2|E|$, the token circulation realized by the standard random walk is unfair. The frequencies of a vertex u enjoying the token is proportional to $d(u)$, and hence on a Lollipop graph for example, a vertex enjoys receiving the token n times as many as another vertex. This is another motivation of introducing a biased random walk. Ikeda et al.[18] proposed the following random walk. As in above, let v be any vertex in $N(u)$. Then u selects v with the probability $\min\{d^{-1}(u), d^{-1}(v)\}$. Note that u selects u itself with the probability $1 - \sum_{v \in N(u)} \min\{d^{-1}(u), d^{-1}(v)\}$. Then the transition probability matrix is now doubly stochastic, and the corresponding stationary distribution becomes uniform – the corresponding token circulation becomes fair.

That all processes enjoy the fair token circulation is a good news. However, this random walk introduces self-loops, which may cause longer hitting and/or cover times. It is a good question to ask if there is a biased random walk such that it achieves a given probability distribution as its stationary distribution and also achieves faster hitting and cover times. We answer this question in the next section.

The Israeli and Jalfon's algorithm assumes that the communication network does not change, but the communication networks dynamically change in many distributed systems. Hence random walks on dynamic graphs are worth discussing, although such random walks had not been investigated until recently even for the standard random walks. In the next section, we also discuss random walks on dynamic graphs. Obviously we are interested in biased random walks (not only stationary ones) on dynamic graphs, but we can only find results about biased random walks on static graphs or about standard random walks on dynamic graphs, except one.

Performance of a probabilistic self-stabilizing system under the randomized scheduler can also be analyzed using random walks. In Subsection A, we modeled the distributed system $\mathcal{S}$ under the randomized scheduler as a Markov chain $\mathcal{M}$ with a transition probability matrix $\mathbf{P} = (p_{\gamma\gamma'})_{\gamma,\gamma' \in \mathcal{C}}$. Without loss of generality, we may assume that any action in an algorithm must update a local variable, which implies that an activation of different set of processes must yield a different configuration. Since $p_{\gamma\gamma'} = 2^{-|U|}$, where U is the set of enabled processes at γ, we assign the same probability $p_{\gamma\gamma'} = 2^{-|U|}$ to each of the neighbors γ'.

An evolution $x_0, x_1, \ldots$ of $\mathcal{M}$ is identified with a random walk on (di)graph $\mathcal{S}$ that obeys $\mathbf{P}$, when we regard the current state as the token. By definition the random walk obeying $\mathbf{P}$ is the standard random walk.[3] Provided that $\mathcal{S}$ is finite state, $H_{\mathcal{S}}$ (and hence $C_{\mathcal{S}}$) bounds the *convergence time* of probabilistic self-stabilizing system $\mathcal{S}$, where the convergence time is the average number of steps necessary for $\mathcal{S}$ to reach a legitimate configuration (starting from the worst configuration). Besides the hitting and cover times, [9, 18] observed that many other properties of random walks like coalescence time, mixing time and blanket time[4] may also affect the performance of a probabilistic self-stabilizing system.

3 Biased Random Walks on Dynamic Graphs

3.1 Biased Random Walks on Static Graphs

We consider random walks on undirected graphs $G = (V, E)$. The standard random walk is *unbiased*, since it always selects one of the neighbors uniformly at random. The other random walks are *biased*. The standard random walk is very popular since its implementation in a distributed system is easy;

[3]Strictly speaking, a standard random walk does not contain a self-loop, i.e., $\gamma \neq \gamma'$. A random walk including a self-loop is sometimes called a *lazy chain*, which is frequently used to modify a periodic Markov chain to make it aperiodic, without changing its fundamental (asymptotic) properties.

[4]For their definitions, see a standard textbook, e.g., [1].

a vertex (i.e., process) u only needs its degree $d(u)$ in G. However, it is also obvious that we can improve the performance of a random walk by using more information.

As touched in Section II, we can design a biased random walk whose hitting and cover times are better than the standard random walk, when $G = (V, E)$ is available for a process u to calculate the transition probability p_{uv} from u to each neighbor v. Let $T = (V, E_T)$ be a spanning tree of G. For any $u, v \in V$, let

$$p_{uv} = \begin{cases} \frac{1}{d_T(u)} & \text{if } (u,v) \in E_T \\ 0 & \text{otherwise,} \end{cases}$$

where $d_T(u)$ is the degree of u in T. Then the token circulates in T. Since both the cover and the hitting times of the standard random walk on a tree are $O(n^2)$ [2], both the cover and the hitting times are $O(n^2)$ for any graph G with order n, which improve the matching bound $\Theta(n^3)$ on the cover and the hitting times of standard random walk by a factor of n. Since both the cover and the hitting times of a path are $\Omega(n^2)$ for any transition probability matrix $\mathbf{P}$ [19], this biased random walk is indeed best possible. The hitting and the cover times do not match in general. The Mattheus's lemma [23] relates them: For any $G = (V, E)$ with order n and transition probability matrix $\mathbf{P}$,

$$h_{n-1} \min_{u,v \in V, u \neq v} H_G(u,v) \leq C_G \leq h_{n-1} \max_{u,v \in V, u \neq v} H_G(u,v),$$

where h_n denotes the n-th harmonic number. From the view of the design of a good distributed algorithm, however, a global information like G is usually not available for a process.

We therefore focus on biased random walks that use only local information. Ref. [19] proposes the β-random walk, whose transition probability is defined by

$$p_{uv} = \begin{cases} \frac{d^{-\beta}(v)}{\sum_{w \in N(u)} d^{-\beta}(w)} & \text{if } v \in N(u) \\ 0 & \text{otherwise,} \end{cases}$$

where β is a constant, and shows that it exhibits a good performance when $\beta = 1/2$.

Theorem 8 ([19]). *For any graph of order n, the hitting time of $1/2$-random walk is $O(n^2)$, which is best possible as observed, and its cover time is $O(n^2 \log n)$, for which there is still a gap from the $\Omega(n^2)$ bound.*

Given a finite graph $G = (V, E)$ and a probability distribution $\pi = (\pi_v)_{v \in V}$ on V, the Metropolis walk is defined as a random walk whose transition probability is defined by

$$p_{uv} = \begin{cases} \frac{1}{d(u)} \min\{\frac{d(u)\pi_v}{d(v)\pi_u}, 1\} & \text{if } v \in N(u) \\ 1 - \sum_{w \neq u} p_{uw} & \text{if } u = v \\ 0 & \text{otherwise,} \end{cases}$$

and guarantees to have π as its stationary distribution [16, 24]. The Metropolis walk has been used frequently in the context of random sampling named Markov chain Monte Carlo (MCMC). Let $f = \max_{u,v \in V} \pi_u / \pi_v$.

Theorem 9 ([27]). *The hitting time of the Metropolis walk is $\Theta(fn^2)$, and its cover time is $\Theta(fn^2 \log n)$.*

As an application, if we take π to be uniform and use the Metropolis walk in the Israeli and Jalfon's algorithm, instead of the standard random walk, a uniform token circulation algorithm with a smaller hitting and cover times is derived, whereas the stationary distribution of standard random walk at u is $d(u)/2|E|$, which is proportional to its degree and is not uniform.

Confirm that both the β-random walk and the Metropolis walk make use of local information, the degrees of the neighbors and itself.

Since information on a different process is expensive to obtain in any case, it is interesting to estimate the goodness of standard random walk, compared with any biased random walk. In general, the ratio between the hitting (and cover) time of standard random walk to the fastest one is $\Theta(n^2)$, since there is a Hamiltonian graph G such that the hitting and the cover times of standard random walk on G are $\Omega(n^3)$, and that there is a random walk designed for G that deterministically circulates the token along the Hamiltonian circuit of G (and thus its hitting and the cover times are n). The next theorem shows that the standard random walk is considerably good as long as we use it on a tree.

Theorem 10 ([28]). *On any tree T, the ratio between the hitting time (resp. the cover time) of standard random walk on T to the fastest one (designed for T) is $O(\sqrt{n})$ (resp. $O(\sqrt{n}\log n)$).*

3.2 Standard Random Walks on Dynamic Graphs

In this subsection, we discuss random walks on dynamic graphs $\mathcal{G} = \{G_t = (V, E_t) : t \in \mathbf{N}\}$, where G_t is the graph at time t. Thus during the token circulation, the vertex set does not change and only the edge set changes. Although dynamic graphs are difficult to handle in general, some cases are easy to analyze. Suppose that for any t, we can construct G_t by taking each pair $\{u, v\}$ of distinct vertices in V at random with the fixed probability $p(0 < p < 1)$. Then the standard random walk works very well and the hitting time is $O(n)$ and the cover time is $O(n \log n)$, i.e., they are the same as those on the complete graph. Let $G = (V, E)$ be a connected graph. If we restrict the above graph construction procedure so that an edge is selected from a set E, the hitting and the cover times of standard random walk on $\mathcal{G}$ are $O(H_G)$ and $O(C_G)$. In [21], the authors extended these observations to estimate the hitting and the cover times of $\beta-$random walks on dynamic graphs, for two models; choosing destination before knowing the new incident edges and choosing destination after knowing them.

Let $\mathbf{G}$ is the set of all connected graphs on the vertex set V. $\mathcal{G}$ is Markovian evolving, if $\mathcal{G}$ is a Markov chain whose state set is $\mathbf{G}$. Avin et al. [5] discussed the case in which $\mathcal{G}$ is Markovian evolving. As you might expect, there are Markovian evolving graphs on which the standard random walk does not perform well.

Theorem 11 ([5]). *There is a Markovian evolving graph $\mathcal{G}$ such that the hitting time of the standard walk is $\Omega(2^n)$. Also there is a Markovian evolving graph $\mathcal{G}$ such that the cover time of the standard walk is $O(2^{\Omega(n^{\epsilon})})$, for any $0 < \epsilon < 1$ and $n \geq 2^{1/(1-\epsilon)}$.*

However, if we restrict $\mathcal{G}$ to be Bernoulli in the sense that G_1 and G_i are taken at random according to a fixed probability distribution from $\mathbf{G}$, then we can obtain polynomial time results:

Theorem 12 ([5]). *The hitting time (resp. the cover time) of the standard random walk on any Bernoulli evolving graph is $O(n^3)$ (resp. $O(n^3 \log n)$).*

4 Discussions and Conclusions

In this paper, we have introduced the concept of probabilistic self-stabilization, in the context of reducing the difficulty of designing a self-stabilizing algorithm. Since a weak stabilizing algorithm is self-stabilizing when the scheduler satisfies the Goulda's strong fairness, and a weak stabilizing system is substantially easier to design than a self-stabilizing system, designing a weak stabilizing algorithm and running it under a scheduler that observes the Goulda's strong fairness are a systematic approach to design a self-stabilizing system. An drawback of the approach however is that the Goulda's strongly fairness is substantially stronger than the strong fairness (which we may naturally expect to the environment), and it may be difficult for us to control the scheduler so that it observes the Goulda's strongly fairness, since the scheduler is an abstraction of the environment.

The environment in general contains much uncertainty and one may wish to model it as a stochastic process. The results in Section II verified an advantage of the model; roughly a weak stabilizing algorithm is always probabilistic self-stabilizing under a randomized scheduler, as long as its state space is finite.

Another advantage of introducing a randomized scheduler is that many problems become solvable in a probabilistic self-stabilizing manner on anonymous networks, on which many problems are unsolvable by deterministic systems. As discussed in Subsection C of Section II, we could design probabilistic self-stabilizing algorithms to solve some problems on anonymous networks, for which there are no self-stabilizing algorithms. Indeed, many of the probabilistic self-stabilizing algorithms proposed so far are designed to work on anonymous networks.

The probabilistic self-stabilization is also considered under other distributed computing models that are proposed to investigate other classes of distributed systems than traditional distributed computer networks. A probabilistic self-stabilizing algorithm for solving the leader election problem on population protocol is proposed in [4]. The population protocol assumes the *global fairness*, which is a synonym of the Gouda's strong fairness.[5] The protocol can be non-deterministic. By definition the population protocol uses a constant memory, independently of the size of the system. However, to solve the self-stabilizing leader election problem, n states are necessary and sufficient, where n is the number of agents. Thus the problem is unsolvable unless we generalize the population protocol model [8] (see also [25] for a similar result on mediated population protocol).

In [4], the problem is solved when the graph is a ring of an odd size. Since the protocol is finite and the scheduler satisfies the global fairness, they needed to design a weak stabilizing protocol, and indeed they did.

The rendezvous problem for mobile agents on a graph is also unsolvable in general when both mobile agents and ring are anonymous, as observed in [22]. However, there is a simple probabilistic self-stabilizing algorithm; let each agent make the standard random walk. Like the algorithm in Subsection C, one can improve the coalescence time by adopting a biased random walk such as the β-random walk if the graph is not regular.

As a related problem, the geometric pattern formation problem by anonymous mobile robots on a plane has also been discussed (see e.g., [29]). It is shown that any memory-less pattern formation algorithm is self-stabilizing. However, not all patterns are formable by the robots, and even two robots cannot meet in the worst case, under the deterministic setting, mainly because of the impossibility of breaking the symmetry. If we are allowed to use a probabilistic algorithm, it is not hard to construct a probabilistic self-stabilizing algorithm for forming any pattern. Note that the mobile agents and robots model assume the strongly fair scheduler.

Researchers are fascinated by the autonomy (i.e., self-* ness, such as self-organization, self-healing, and self-stabilization) that many natural distributed systems exhibit, despite of their anonymity, and ask why? Since the notion of probabilistic self-stabilization works very well in anonymous setting as we have observed in this section, it is a natural guess that the probabilistic self-stabilization plays the central role in natural autonomous systems, particularly when the environment can be modeled as a randomized scheduler. Thus it may be interesting to view the natural computation from the view of probabilistic self-stabilization.

In Section II, we introduced the randomness in two ways. Although the behavior of a randomized algorithm under a randomized scheduler can be analyzed by using a Markov chain, the roles of the two randomness may be different; we cannot control a given randomized scheduler, but we can design a randomized algorithm. It may be interesting to consider how to design a good randomized algorithm for a given randomized scheduler.

In Section III we explained some techniques to improve the performance of random walks, after explaining the relation between the probabilistic self-stabilization and the Markov chain. Although some researchers observed this relation and analyzed a probabilistic self-stabilizing algorithm using the Markov chain corresponding to it (e.g., [26]), designing a good random walk for a distributed algorithm is still left open. Ref. [9] surveys a variety of issues. For example, multiple random walks that simultaneously circulating multiple tokens to reduce the cover time [3].

References

[1] D.J. Aldous and J. Fill. Reversible Markov Chains and Random Walks on Graphs. http://www.stat.berkeley.edu/users/aldous/RWG/book.html (monograph in preparation).

[5] The *local fairness* is a synonym of the strong fairness.

[2] R. Aleliunas, R.M Karp, R.J. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. *Proc. 20th Ann. Symposium on Foundations of Computer Science*, 218–223, 1979.

[3] N. Alon, C. Avin, M. Koucký, G. Kozma, Z. Lotker, and M. Tuttle. Many random walks are faster than one. *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures,* 119–129, 2008.

[4] D. Angluin, J. Aspnes, M. J Fisher, and H. Jiang. Stabilizing population protocols. *ACM Trans. Autonomous and Adaptive Systems,* 3, 4, 13, 2008.

[5] C. Avin, M. Koucký, and Z. Lotker. How to explore a fast-changing world − Cover time of a simple random walk on evolving graphs. *Proc. 35th International Colloquium on Automata, Languages and Programming,* 121–132, 2008.

[6] J. E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Trans. Programming Languages and Systems* 11, 2, 330–340, 1989.

[7] P. Billingsley. Probability and Measure. *Wiley Series in Probability and Mathematical Statistics*, John Wiley & Sons, Inc., New York, 1995.

[8] S. Cai, T. Izumi, and K. Wada. Space complexity of self-stabilizing leader election in passively-mobile anonymous agents. *Proc. 16th Colloquium on Structure, Information, Communication, and Complexity,* 113–125, 2009.

[9] C. Cooper. Random walks, interacting particles, dynamic networks: Randomness can Be helpful. *Proc. 18th Colloquium on Structure, Information, Communication, and Complexity,* 1–14, 2011.

[10] S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. Self vs. Probabilistic Stabilization. *Proc. 28th IEEE International Conference on Distributed Computing Systems,* 681–688, 2008.

[11] S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. Self vs. Probabilistic Stabilization, (preparing for submission).

[12] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17, 11, 643–644, 1974.

[13] U. Feige. A tight upper bound on the cover time for random walks on graphs. *J. Random Structures and Algorithms,* 6, 51–54, 1995.

[14] U. Feige, A tight lower bound on the cover time for random walks on graphs. *J. Random Structures and Algorithms,* 6, 433–438, 1995.

[15] Mohamed G. Gouda. The theory of weak stabilization. *Proc. 5th International Workshop on Self-Stabilizing Systems,* 114–123, 2001.

[16] W.K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika,* 57, 97–109, 1970.

[17] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35,2, 63–67, 1990.

[18] S. Ikeda, I. Kubo, I. Okumoto, and M. Yamashita. Fair circulation of a token. *IEEE Trans. Parallel and Distributed Systems,* 13, 4, 367–372, 2002.

[19] S. Ikeda, I. Kubo, and M. Yamashita. The hitting and the cover times of random walks on finite graphs using local degree information. *Theoretical Computer Science,* 410, 94-100, 2009.

[20] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. *Proc. 9th ACM Symposium on Principles of Distributed Computing*, 119–131, 1990.

[21] K. Koba, S. Kijima and M. Yamashita. Random walks on dynamic graphs. preprint 2010 (in Japanese)

[22] E. Kranakis, N. Santoro, C. Sawchuk, and D. Krizanc. Mobile agent rendezvous in a ring. *Proc. 23rd International Conference on Distributed Computing Systems,* 592, 2003.

[23] P. Matthews. Covering problems for markov chain. *The Annals of Probability,* 16, 3, 1215-1228, 1988.

[24] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equations of state calculations by fast computing machine. *Journal of Chemical Physics,* 21, 1087–1091, 1953.

[25] R. Mizoguchi, H. Ono, S. Kijima, and M. Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing,* 2012 (to appear).

[26] T. Nakata. On the expected time for Herman's probabilistic self-stabilizing algorithm. *Theoretical Computer Science,* 349, 475–483, 2005.

[27] Y. Nonaka, H. Ono, K. Sadakane, and M. Yamashita. The Hitting and Cover Times of Metropolis Walks. *Theoretical Computer Science,* 411(16-18), 1889–1894, 2010.

[28] Y. Nonaka, H. Ono, S. Kijima, and M. Yamashita. How Slow, or Fast, are Standard Random Walks? – Analysis of Hitting and Cover Times on Trees. *The 17th Computing: The Australasian Theory Symposium (CATS 2011),* Perth, Australia, January 2011.

[29] I. Suzuki and M. Yamashita. A Theory of distributed anonymous mobile robots – Formation and agreement problems. *SIAM J. Computing,* 28, 4, 1347–1363, 1999.